mikroPascal PRO for PIC32™

Manual

mikroPascal PRO for PIC32 is a full-featured C compiler for PIC32 MCUs from Microchip. It is designed for developing, building and debugging PIC32-based embedded applications. This development environment has a wide range of features such as: easy-to-use IDE, very compact and efficient code, many hardware and software libraries, comprehensive documentation, software simulator, COFF file generation, SSA optimization (up to 30% code reduction) and many more. Numerous ready-to-use and well-explained examples will give a good start for your embedded project.

Compiler

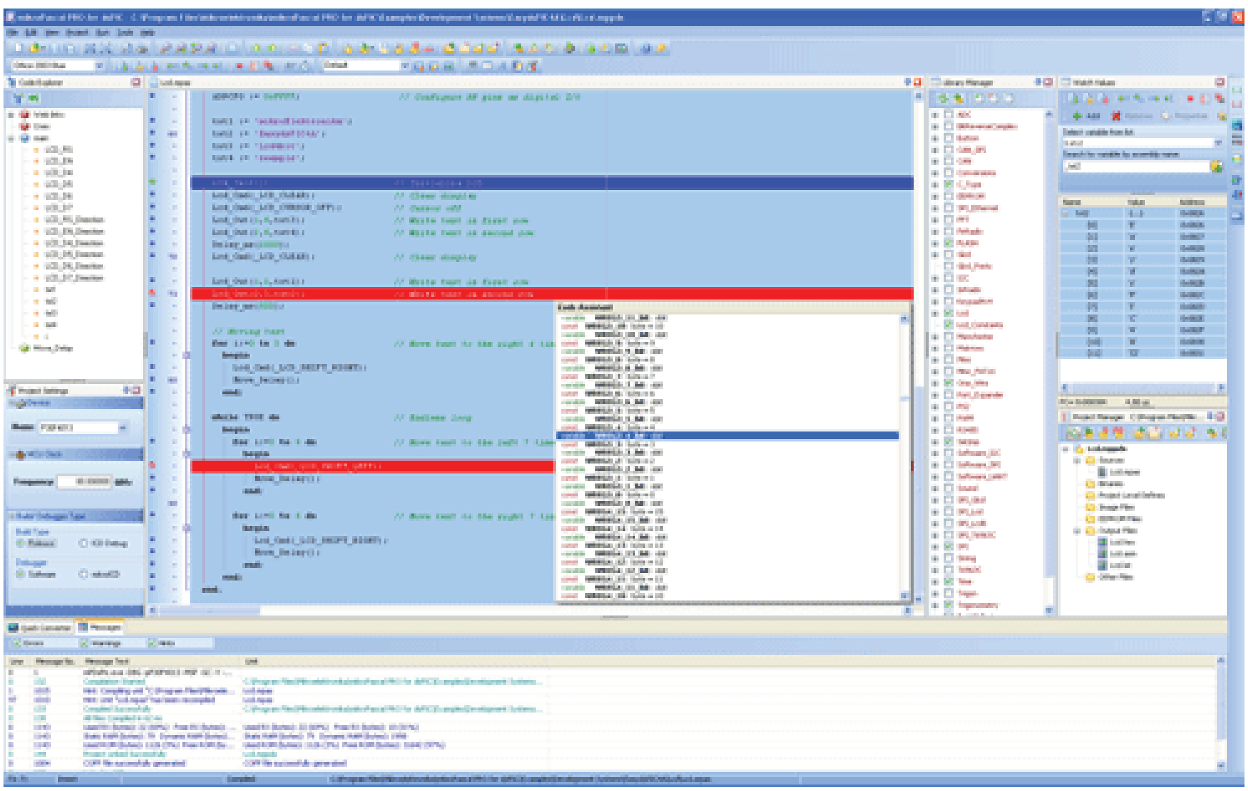
MikroElektronika
SOFTWARE AND HARDWARE SOLUTIONS FOR EMBEDDED WORLD ...making it simple

Table of Contents

CHAPTER 1

INTRODUCTION

The mikroPascal PRO for PIC32 is a powerful, feature-rich development tool for PIC32 microcontrollers. It is designed to provide the programmer with the easiest possible solution to developing applications for embedded systems, without compromising performance or control.



mikroPascal PRO for PIC32 IDE

Introduction to mikroPascal PRO for PIC32

The PIC32 is a 32-bit family of general purpose microcontrollers. This is the Microchip's first inherent 32-bit (data) microcontroller family. It builds upon the MIPS M4K 32-bit core, offering high-performance hardware multiply/divide unit, programmable user and kernel memory partition through an unified 4GB virtual memory space, with powerful peripherals to address a wide range of applications.

Having a wide range of application, being prized for its efficiency, PIC32 MCUs are a natural choice for developing embedded systems. mikroPascal PRO for PIC32 provides a successful match featuring highly advanced IDE, ANSI compliant compiler, broad set of hardware libraries, comprehensive documentation, and plenty of ready-to-run examples.

Features

mikroPascal PRO for PIC32 allows you to quickly develop and deploy complex applications:

- Write your source code using the built-in Code Editor (Code and Parameter Assistants, Code Folding, Syntax Highlighting, Auto Correct, Code Templates, and more.)
- Use included mikroPascal PRO for PIC32 libraries to dramatically speed up the development: data acquisition, memory, displays, conversions, communication etc.
- Monitor your program structure, variables, and functions in the Code Explorer.
- Generate commented, human-readable assembly, and standard HEX compatible with all programmers.
- Use the integrated mikroICD (In-Circuit Debugger) Real-Time debugging tool to monitor program execution on the hardware level.
- Inspect program flow and debug executable logic with the integrated Software Simulator.
- Use Single Static Assignment optimization to shrink your code to even smaller size.
- Get detailed reports and graphs: RAM and ROM map, code statistics, assembly listing, calling tree, and more.
- Active Comments enable you to make your comments alive and interactive.
- mikroPascal PRO for PIC32 provides plenty of examples to expand, develop, and use as building bricks in your projects. Copy them entirely if you deem fit – that's why we included them with the compiler.

Where to Start

- In case that you're a beginner in programming the PIC32 microcontrollers, read carefully the PIC32 Specifics chapter. It might give you some useful pointers on the PIC32 constraints, code portability, and good programming practices.
- If you are experienced in Pascal programming, you will probably want to consult mikroPascal PRO for PIC32 Specifics first. For language issues, you can always refer to the comprehensive Language Reference. A complete list of included libraries is available at mikroPascal PRO for PIC32 Libraries.
- If you are not very experienced in Pascal programming, don't panic! mikroPascal PRO for PIC32 provides plenty of examples making it easy for you to go quickly through it. We suggest that you first consult Projects and Source Files first, and then start browsing the examples that you're the most interested in.

Software License Agreement

mikroElektronika Associates License Statement and Limited Warranty

IMPORTANT - READ CAREFULLY

This license statement and limited warranty constitute a legal agreement (“License Agreement”) between you (either as an individual or a single entity) and mikroElektronika (“mikroElektronika Associates”) for software product (“Software”) identified above, including any software, media, and accompanying on-line or printed documentation.

BY INSTALLING, COPYING, OR OTHERWISE USING SOFTWARE, YOU AGREE TO BE BOUND BY ALL TERMS AND CONDITIONS OF THE LICENSE AGREEMENT.

Upon your acceptance of the terms and conditions of the License Agreement, mikroElektronika Associates grants you the right to use Software in a way provided below.

This Software is owned by mikroElektronika Associates and is protected by copyright law and international copyright treaty. Therefore, you must treat this Software like any other copyright material (e.g., a book).

You may transfer Software and documentation on a permanent basis provided. You retain no copies and the recipient agrees to the terms of the License Agreement. Except as provided in the License Agreement, you may not transfer, rent, lease, lend, copy, modify, translate, sublicense, time-share or electronically transmit or receive Software, media or documentation. You acknowledge that Software in the source code form remains a confidential trade secret of mikroElektronika Associates and therefore you agree not to modify Software or attempt to reverse engineer, decompile, or disassemble it, except and only to the extent that such activity is expressly permitted by applicable law notwithstanding this limitation.

If you have purchased an upgrade version of Software, it constitutes a single product with the mikroElektronika Associates software that you upgraded. You may use the upgrade version of Software only in accordance with the License Agreement.

LIMITED WARRANTY

Respectfully excepting the Redistributables, which are provided “as is”, without warranty of any kind, mikroElektronika Associates warrants that Software, once updated and properly used, will perform substantially in accordance with the accompanying documentation, and Software media will be free from defects in materials and workmanship, for a period of ninety (90) days from the date of receipt. Any implied warranties on Software are limited to ninety (90) days.

mikroElektronika Associates’ and its suppliers’ entire liability and your exclusive remedy shall be, at mikroElektronika Associates’ option, either (a) return of the price paid, or (b) repair or replacement of Software that does not meet mikroElektronika Associates’ Limited Warranty and which is returned to mikroElektronika Associates with a copy of your receipt. DO NOT RETURN ANY PRODUCT UNTIL YOU HAVE CALLED MIKROELEKTRONIKA ASSOCIATES FIRST AND OBTAINED A RETURN AUTHORIZATION NUMBER. This Limited Warranty is void if failure of Software has resulted from an accident, abuse, or misapplication. Any replacement of Software will be warranted for the rest of the original warranty period or thirty (30) days, whichever is longer.

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, MIKROELEKTRONIKA ASSOCIATES AND ITS SUPPLIERS DISCLAIM ALL OTHER WARRANTIES AND CONDITIONS, EITHER EXPRESSED OR IMPLIED, INCLUDED, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NON-INFRINGEMENT, WITH REGARD TO SOFTWARE, AND THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES.

IN NO EVENT SHALL MIKROELEKTRONIKA ASSOCIATES OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS AND BUSINESS INFORMATION, BUSINESS INTERRUPTION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE SOFTWARE PRODUCT OR THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES, EVEN IF MIKROELEKTRONIKA ASSOCIATES HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN ANY CASE, MIKROELEKTRONIKA ASSOCIATES' ENTIRE LIABILITY UNDER ANY PROVISION OF THIS LICENSE AGREEMENT SHALL BE LIMITED TO THE AMOUNT ACTUALLY PAID BY YOU FOR SOFTWARE PRODUCT PROVIDED, HOWEVER, IF YOU HAVE ENTERED INTO A MIKROELEKTRONIKA ASSOCIATES SUPPORT SERVICES AGREEMENT, MIKROELEKTRONIKA ASSOCIATES' ENTIRE LIABILITY REGARDING SUPPORT SERVICES SHALL BE GOVERNED BY THE TERMS OF THAT AGREEMENT.

HIGH RISK ACTIVITIES

Software is not fault-tolerant and is not designed, manufactured or intended for use or resale as on-line control equipment in hazardous environments requiring fail-safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, direct life support machines, or weapons systems, in which the failure of Software could lead directly to death, personal injury, or severe physical or environmental damage ("High Risk Activities"). mikroElektronika Associates and its suppliers specifically disclaim any expressed or implied warranty of fitness for High Risk Activities.

GENERAL PROVISIONS

This statement may only be modified in writing signed by you and an authorised officer of mikroElektronika Associates. If any provision of this statement is found void or unenforceable, the remainder will remain valid and enforceable according to its terms. If any remedy provided is determined to have failed for its essential purpose, all limitations of liability and exclusions of damages set forth in the Limited Warranty shall remain in effect.

This statement gives you specific legal rights; you may have others, which vary, from country to country. mikroElektronika Associates reserves all rights not specifically granted in this statement.

mikroElektronika
Visegradska 1A,
11000 Belgrade,
Europe.

Phone: + 381 11 36 28 830
Fax: +381 11 36 28 831
Web: www.mikroe.com
E-mail: office@mikroe.com

Technical Support

The latest software can be downloaded free of charge via Internet (you might want to bookmark the page so you could check news, patches, and upgrades later on): http://www.pic32compilers.com/ .

In case you encounter any problem, you are welcome to our support forums at www.mikroe.com/forum/. Here, you may also find helpful information, hardware tips, and practical code snippets. Your comments and suggestions on future development of the mikroPascal PRO for PIC32 are always appreciated — feel free to drop a note or two on our Wishlist.

In our Knowledge Base www.mikroe.com/en/kb/ you can find the answers to Frequently Asked Questions and solutions to known problems. If you can not find the solution to your problem in Knowledge Base then report it to Support Desk www.mikroe.com/en/support/. In this way, we can record and track down bugs more efficiently, which is in our mutual interest. We respond to every bug report and question in a suitable manner, ever improving our technical support.

How to Register

The latest version of the mikroPascal PRO for PIC32 is always available for downloading from our website. It is a fully functional software with the mikroICD(in-circuit Debugger), all the libraries, examples, and comprehensive help included.

The only limitation of the free version is that it cannot generate hex output over 2K of program words. Although it might sound restrictive, this margin allows you to develop practical, working applications with no thinking of demo limit. If you intend to develop really complex projects in the mikroPascal PRO for PIC32, then you should consider the possibility of purchasing the license key.

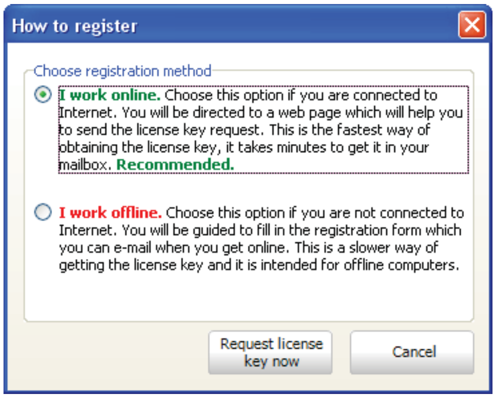
Who Gets the License Key

Buyers of the mikroPascal PRO for PIC32 are entitled to the license key. After you have completed the payment procedure, you have an option of registering your mikroPascal PRO for PIC32. In this way you can generate hex output without any limitations.

How to Get License Key

After you have completed the payment procedure, start the program. Select **Help › How to Register** from the drop-down menu or click the How To Register Icon  .

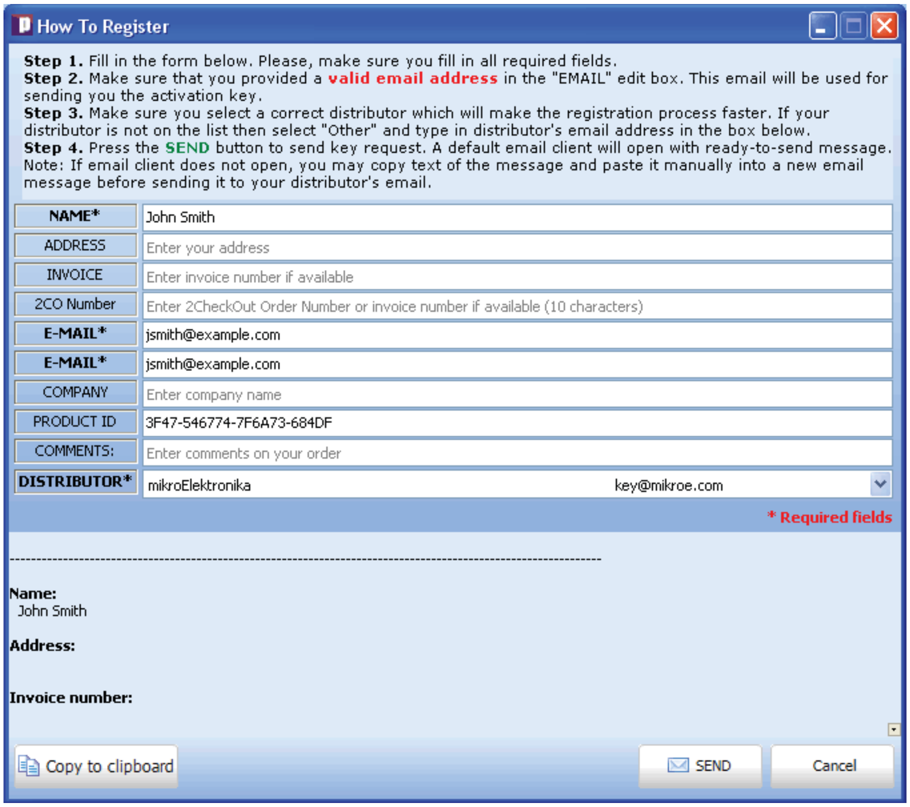
You can choose between two registering methods, **I work online** or **I work offline**, based on your current internet connection and click **Request license key now** button:



If you choose **I work online** registering method, following page will be opened in your default browser:

Fill out the registration form, select your distributor, and click the **Submit** button.

If you choose **I work offline** registering method, following window will be opened:



Fill out the registration form, select your distributor, and click the **Submit** button.

This will start your e-mail client with message ready for sending. Review the information you have entered, and add the comment if you deem it necessary. Please, do not modify the subject line.

Upon receiving and verifying your request, we will send the license key to the e-mail address you specified in the form.

After Receving the License Key

The license key comes as a small autoextracting file – just start it anywhere on your computer in order to activate your copy of compiler and remove the demo limit. You do not need to restart your computer or install any additional components. Also, there is no need to run the mikroPascal PRO for PIC32 at the time of activation.

Important:

- The license key is valid until you format your hard disk. In case you need to format the hard disk, you should request a new activation key.
- Please keep the activation program in a safe place. Every time you upgrade the compiler you should start this program again in order to reactivate the license.

CHAPTER 2

mikroPascal PRO for PIC32 Environment

Main Menu Options

Available Main Menu options are:

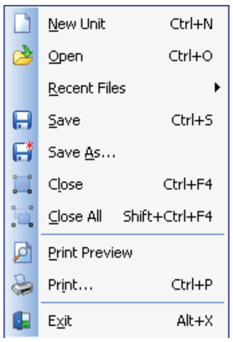
- File
- Edit
- View
- Project
- Build
- Run
- Tools
- Help

Related topics: Keyboard shortcuts, Toolbars

File

File Menu Options

The File menu is the main entry point for manipulation with the source files.



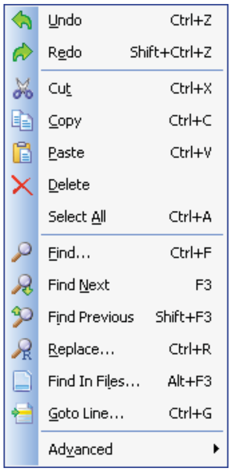
File	Description
New Unit Ctrl+N	Open a new editor window.
Open Ctrl+O	Open source file for editing or image file for viewing.
Recent Files	Reopen recently used file.
Save Ctrl+S	Save changes for active editor.
Save As...	Save the active source file with the different name or change the file type.
Close Ctrl+F4	Close active source file.
Close All Shift+Ctrl+F4	Close all opened files.
Print Preview	Print Preview.
Print... Ctrl+P	Print.
Exit Alt+X	Exit IDE.

Related topics: Keyboard shortcuts, File Toolbar, Managing Source Files

Edit

Edit Menu Options

The Edit Menu contains commands for editing the contents of the current document.

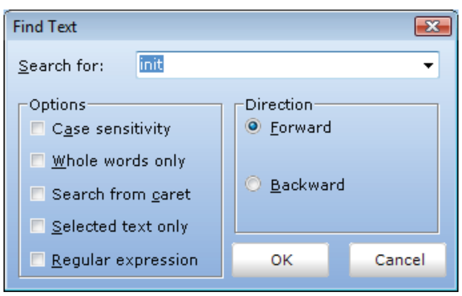


Edit	Description
Undo Ctrl+Z	Undo last change.
Redo Shift+Ctrl+Z	Redo last change.
Cut Ctrl+X	Cut selected text to clipboard.
Copy Ctrl+C	Copy selected text to clipboard.
Paste Ctrl+V	Paste text from clipboard.
Delete	Delete selected text.
Select All Ctrl+A	Select all text in active editor.
Find... Ctrl+F	Find text in active editor.
Find Next F3	Find next occurence of text in active editor.
Find Previous Shift+F3	Find previous occurence of text in active editor.
Replace... Ctrl+R	Replace text in active editor.
Find In Files... Alt+F3	Find text in current file, in all opened files, or in files from desired folder.
Goto Line... Ctrl+G	Go to line to the desired line in active editor.
Advanced	Advanced Code Editor options

Advanced »	Description
Comment Shift+Ctrl+.	Comment selected code or put single line comment if there is no selection.
Uncomment Shift+Ctrl+,	Uncomment selected code or remove single line comment if there is no selection.
Indent Shift+Ctrl+I	Indent selected code.
Outdent Shift+Ctrl+U	Outdent selected code.
Lowercase Ctrl+Alt+L	Changes selected text case to lowercase.
Uppercase Ctrl+Alt+U	Changes selected text case to uppercase.
Titlecase Ctrl+Alt+T	Changes selected text case to titlercase.

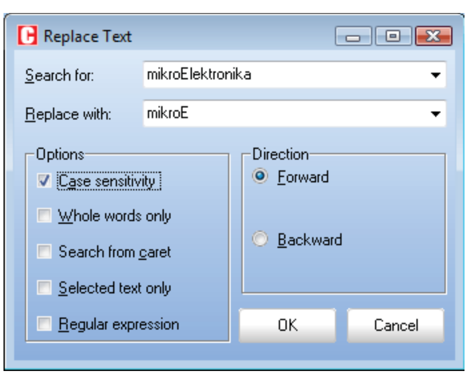
Find Text

Dialog box for searching the document for the specified text. The search is performed in the direction specified. If the string is not found a message is displayed.



Replace Text

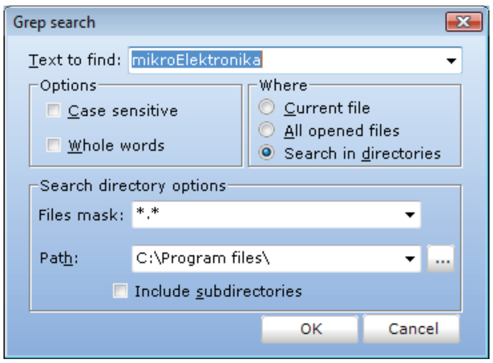
Dialog box for searching for a text string in file and replacing it with another text string.



Find In Files

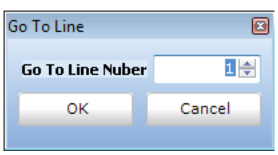
Dialog box for searching for a text string in current file, all opened files, or in files on a disk.

The string to search for is specified in the **Text to find** field. If Search in directories option is selected, The files to search are specified in the **Files mask** and **Path** fields.



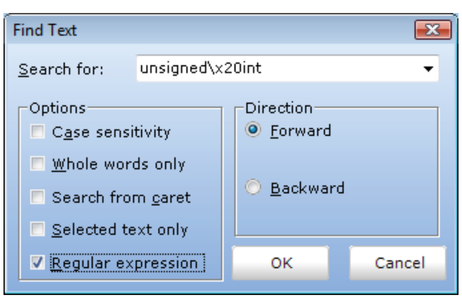
Go To Line

Dialog box that allows the user to specify the line number at which the cursor should be positioned.



Regular expressions option

By checking this box, you will be able to advance your search, through Regular expressions.

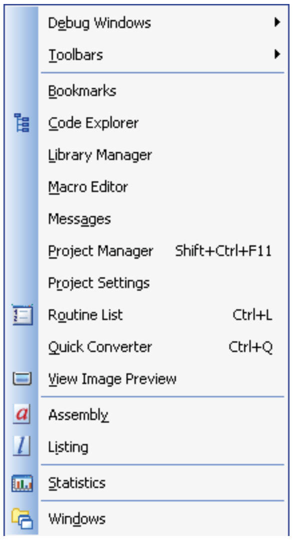


Related topics: Keyboard shortcuts, Edit Toolbar, Advanced Edit Toolbar

View

View Menu Options

View Menu contains commands for controlling the on-screen display of the current project.



View	Description
Debug Windows	Show/Hide Software Simulator / mikroICD (In-Circuit Debugger) Debug Windows.
Toolbars	Show/Hide Toolbars.
Bookmarks	Show/Hide Bookmarks window.
Code Explorer	Show/Hide Code Explorer window.
Library Manager	Show/Hide Library Manager window.
Macro Editor	Show/Hide Macro Editor window.
Messages	Show/Hide Messages window.
Project Manager Shift+Ctrl+F11	Show/Hide Project Manager window.
Project Settings	Show/Hide Project Settings window.
Routine List Ctrl+L	Show/Hide Routine List in active editor.
Quick Converter Ctrl+Q	Show/Hide Quick Converter window.
View Image Preview	Show/Hide View Image Preview window.
View Assembly	View Assembly.
View Listing	View Listing.
View Statistics	View Statistics.
Windows	Show Window List window.

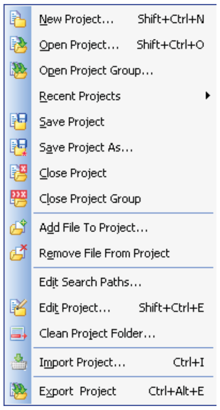
The Tools toolbar can easily be customized by adding new tools in Options(F12) window.

Related topics: Keyboard shortcuts, Integrated Tools

Project

Project Menu Options

Project Menu allows the user to easily manipulate current project.



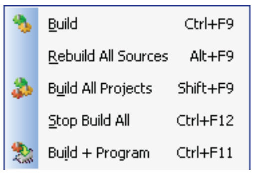
Project	Description
New Project... Shift+Ctrl+N	Open New Project Wizard
Open Project... Shift+Ctrl+O	Open existing project.
Open Project Group...	Open project group.
Recent Projects	Open recently used project or project group.
Save Project	Save current project.
Save Project As...	Save active project file with the different name.
Close Project	Close active project.
Close Project Group	Close project group.
Add File To Project...	Add file to project.
Remove File From Project	Remove file from project.
Edit Search Paths...	Edit search paths.
Edit Project... Shift+Ctrl+E	Edit project settings
Clean Project Folder...	Clean Project Folder
Import Project... Ctrl+I	Import projects created in previous versions of mikroPascal.
Export Project Ctrl+Alt+E	Export Project.

Related topics: Keyboard shortcuts, Project Toolbar, Creating New Project, Project Manager, Project Settings

Build

Build Menu Options

Build Menu allows the user to easily manage building and compiling process.



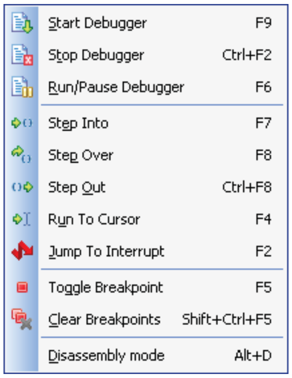
Build	Description
Build Ctrl+F9	Build active project.
Rebuild All Sources Alt+F9	Rebuild all sources in active project.
Build All Projects Shift+F9	Build all projects.
Stop Build All Ctrl+F12	Stop building all projects.
Build + Program Ctrl+F11	Build and program active project.

Related topics: Keyboard shortcuts, Project Toolbar, Creating New Project, Project Manager, Project Settings

Run

Run Menu Options

Run Menu is used to debug and test compiled code on a software or harware level.



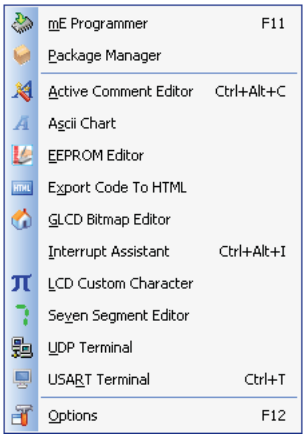
Run	Description
Start Debugger F9	Start Software Simulator or mikroICD (In-Circuit Debugger).
Stop Debugger Ctrl+F2	Stop Debugger.
Run/Pause Debugger F6	Run/Pause Debugger.
Step Into F7	Step Into.
Step Over F8	Step Over.
Step Out Ctrl+F8	Step Out.
Run To Cursor F4	Run To Cursor.
Jump To Interrupt F2	Jump to interrupt in current project.
Toggle Breakpoint F5	Toggle Breakpoint.
Clear Breakpoints Shift+Ctrl+F5	Clear Breakpoints.
Disassembly mode Alt+D	Toggle between source and disassembly.

Related topics: Keyboard shortcuts, Debug Toolbar

Tools

Tools Menu Options

Tools Menu contains a number of applications designed to ease the use of compiler and included library routines.

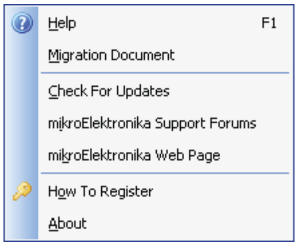


Tools	Description
mE Programmer F11	Run mikroElektronika Programmer.
Package Manager	Run Package Manager.
Active Comment Editor Ctrl+Alt+C	Show/Hide Active Comment Editor window.
Ascii Chart	Run ASCII Chart
EEPROM Editor	Run EEPROM Editor
Export Code To HTML	Generate HTML code suitable for publishing source code on the web.
GLCD Bitmap Editor	Run Glcd bitmap editor
HID Terminal	Run HID Terminal
Interrupt Assistant Ctrl+Alt+I	Run Interrupt Assistant
LCD Custom Character	Run Lcd custom character
Seven Segment Editor	Run Seven Segment Editor
UDP Terminal	Run UDP communication terminal
USART Terminal Ctrl+T	Run USART Terminal
Options F12	Open Options window

Related topics: Keyboard shortcuts, Tools Toolbar

Help

Help Menu Options



Help	Description
Help F1	Open Help File.
Migration Document	Open Code Migration Document.
Check For Updates	Check if new compiler version is available.
mikroElektronika Support Forums	Open mikroElektronika Support Forums in a default browser.
mikroElektronika Web Page	Open mikroElektronika Web Page in a default browser.
How To Register	Information on how to register
About	Open About window.

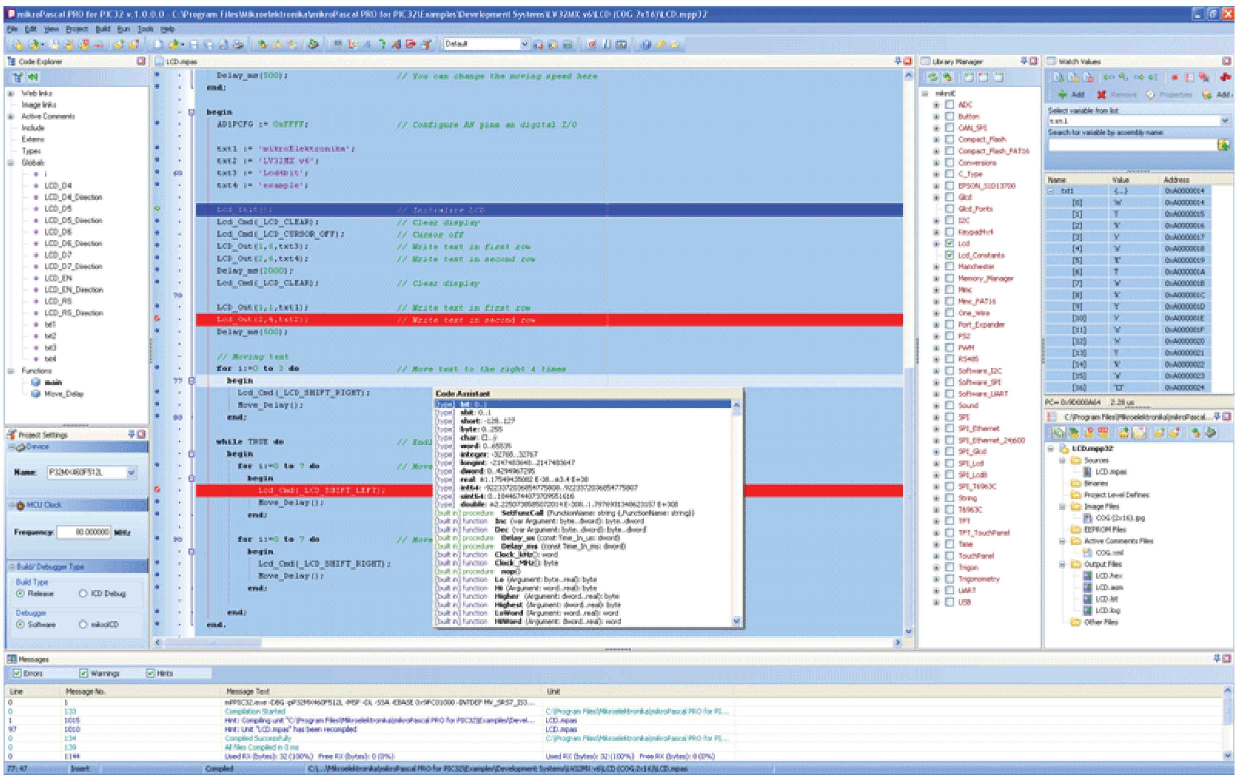
Related topics: Keyboard shortcuts, Help Toolbar

mikroPascal PRO for PIC32 IDE

IDE Overview

The mikroPascal PRO for PIC32 is an user-friendly and intuitive environment.

For a detailed information on a certain part of IDE, simply click on it (hovering a mouse cursor above a desired IDE part will pop-up its name):



- The Code Editor features adjustable Syntax Highlighting, Code Folding, Code Assistant, Parameters Assistant, Spell Checker, Auto Correct for common typos and Code Templates (Auto Complete).
- The Code Explorer is at your disposal for easier project management.
- The Project Manager allows multiple project management
- General project settings can be made in the Project Settings window
- Library manager enables simple handling libraries being used in a project
- The Messages Window displays all messages during compiling and linking.
- The source-level Software Simulator lets you debug executable logic step-by-step by watching the program flow.
- The New Project Wizard is a fast, reliable, and easy way to create a project.
- Help files are syntax and context sensitive.
- Like in any modern Windows application, you may customize the layout of mikroPascal PRO for PIC32 to suit your needs best.
- Spell checker underlines identifiers which are unknown to the project. In this way it helps the programmer to spot potential problems early, much before the project is compiled.
- Spell checker can be disabled by choosing the option in the Preferences dialog (F12).

Code Editor

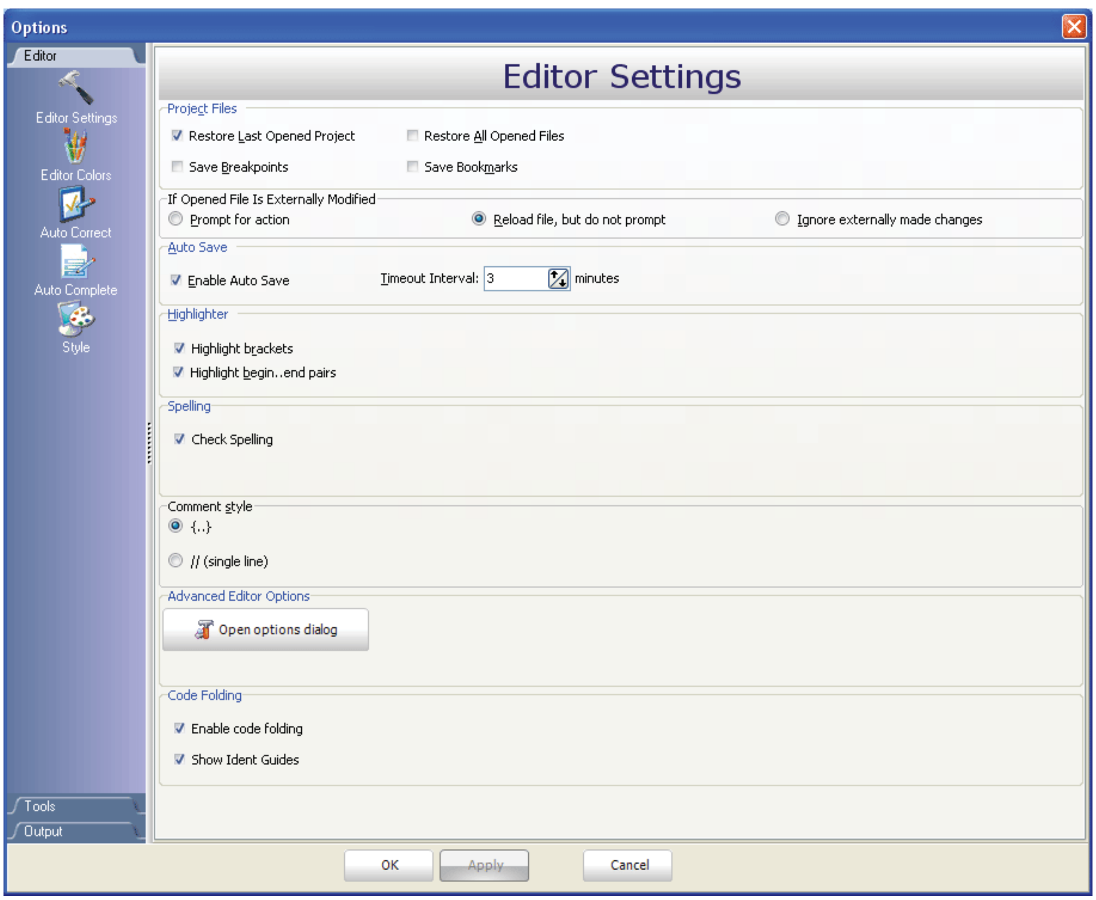
The Code Editor is advanced text editor fashioned to satisfy needs of professionals. General code editing is the same as working with any standard text-editor, including familiar Copy, Paste and Undo actions, common for Windows environment.

Available Code Editor options are: Editor Settings, Editor Colors, Auto Correct, Auto Complete and Style.

Editor Settings

Main Editor Settings Features are:

- Auto Save
- Highlighter
- Spelling
- Comment Style
- Code Folding
- Code Assistant
- Parameter Assistant
- Bookmarks and Go to Line



Auto Save

Auto Save is a function which saves an opened project automatically, helping to reduce the risk of data loss in case of a crash or freeze. Autosaving is done in time intervals defined by the user.

Highlighter

Highlighting is a convenient feature for spotting brackets which notate begin or end of a routine, by making them visually distinct.

Spelling

The Spell Checker underlines unknown objects in the code, so they can be easily noticed and corrected before compiling your project.

Select **Tools › Options** from the drop-down menu, or click the Show Options Icon and then select the Spell Checker Tab.

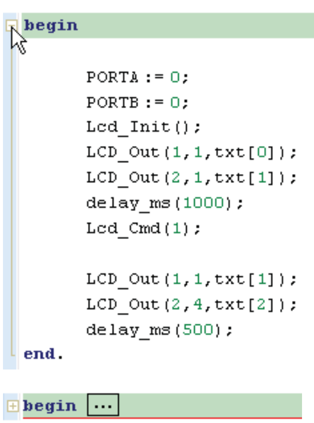
Comment Style

Code Editor has a feature to change the comment style to either single-line or multi-line. Commenting or uncommenting the selected code is done by a simple click of a mouse, using the Comment Icon and Uncomment Icon from the Advanced Edit Toolbar.

Code Folding

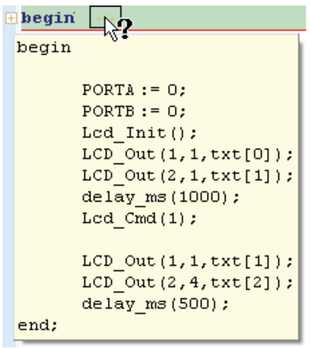
Code folding is IDE feature which allows users to selectively hide and display sections of a source file. In this way it is easier to manage large regions of code within one window, while still viewing only those subsections of the code that are relevant during a particular editing session.

While typing, the code folding symbols (and) appear automatically. Use the folding symbols to hide/unhide the code subsections.



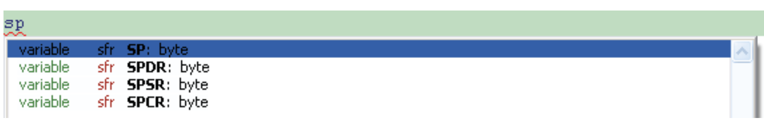
Another way of folding/unfolding code subsections is by using Alt+← and Alt+→.

If you place a mouse cursor over the tooltip box, the collapsed text will be shown in a tooltip style box.



Code Assistant

If you type the first few letters of a word and then press Ctrl+Space, all valid identifiers matching the letters you have typed will be prompted in a floating panel (see the image below). Now you can keep typing to narrow the choice, or you can select one from the list using the keyboard arrows and Enter.



Parameter Assistant

The Parameter Assistant will be automatically invoked when you open parenthesis "(" or press Shift+Ctrl+Space. If the name of a valid function precedes the parenthesis, then the expected parameters will be displayed in a floating panel. As you type the actual parameter, the next expected parameter will become bold.

Bookmarks

Bookmarks make navigation through a large code easier. To set a bookmark, use Ctrl+Shift+*number*. The same principle applies to the removal of the bookmarks. To jump to a bookmark, use Ctrl+*number*.

Go to Line

The Go to Line option makes navigation through a large code easier. Use the shortcut Ctrl+G to activate this option.

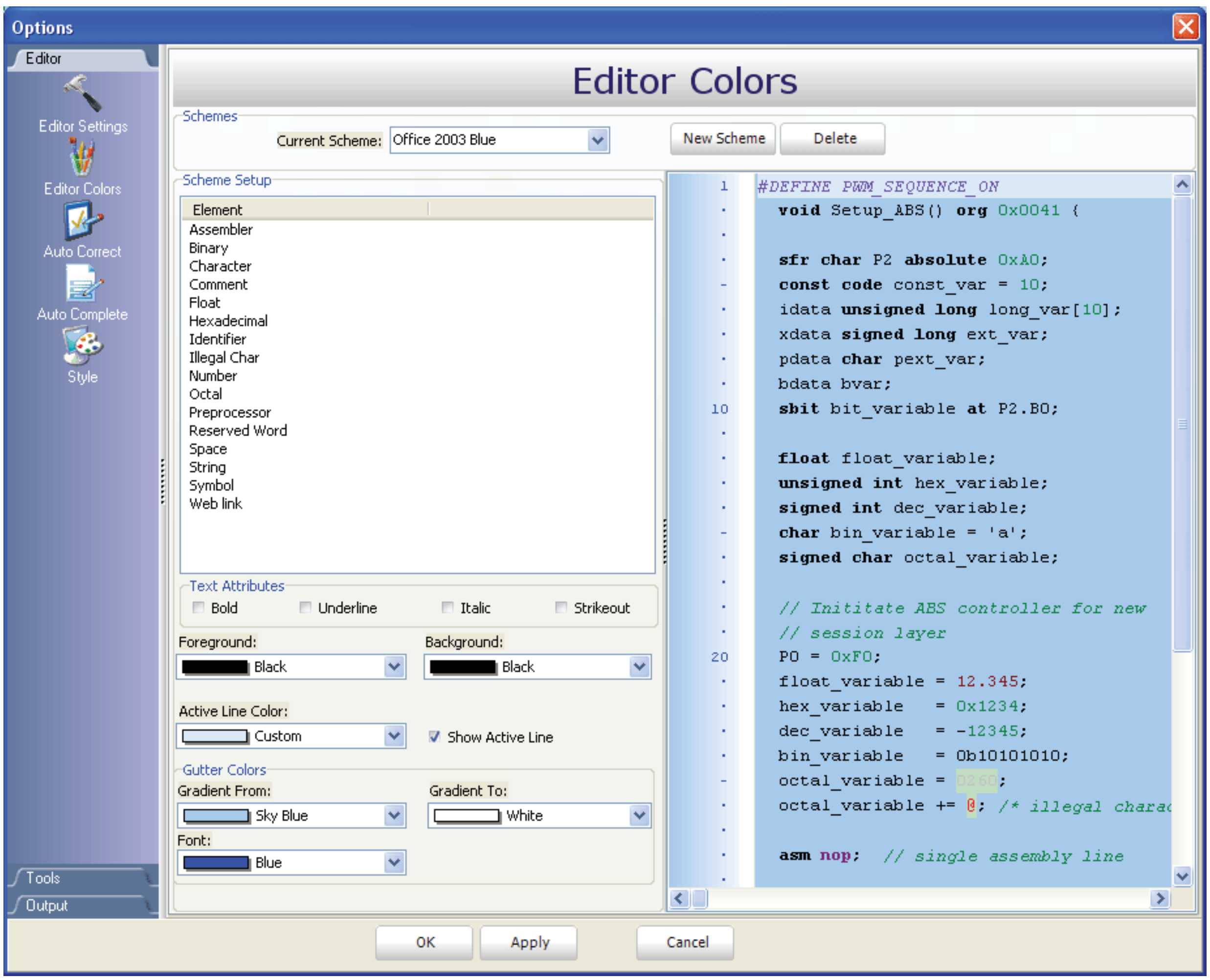
Column Select Mode

This mode changes the operation of the editor for selecting text. When column select mode is used, highlighted text is based on the character column position of the first character selected to the column of the last character of text selected.
Text selected in this mode does not automatically include all text between the start and end position, but includes all text in the columns between the first and last character selected.

Column mode editing is sometimes referred to as block mode editing as the act of selecting text forms a rectangle.

To enter this mode, press Alt + Left mouse button, drag the mouse towards the desired direction thus selecting the text.

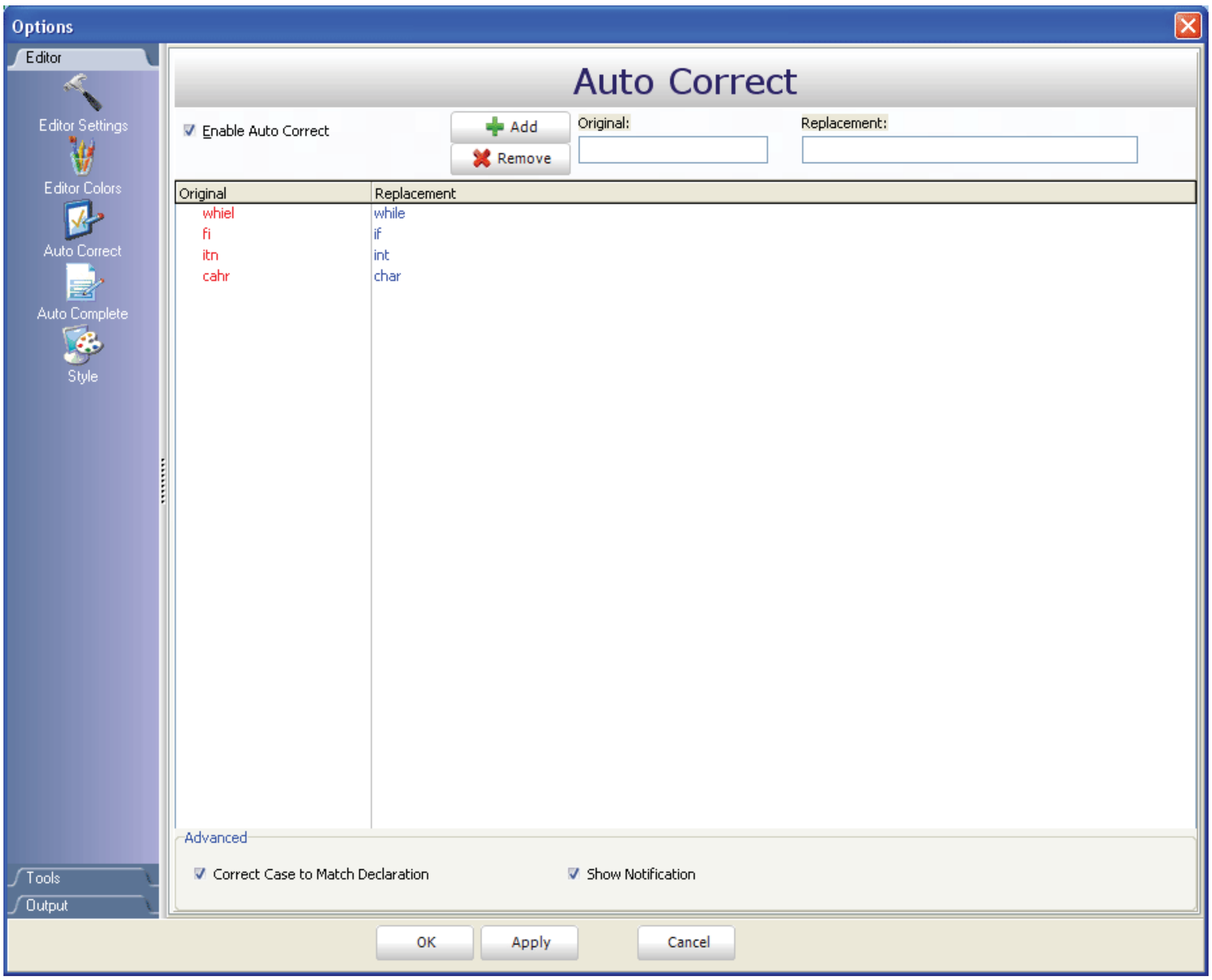
Editor Colors



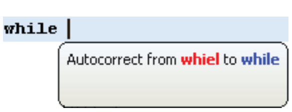
Editor Colors option allows user to set, change and save text and color settings organized in schemes. Schemes represent custom graphical appearance that can be applied to GUI (Graphical User Interface) to satisfy tastes of different users.

Auto Correct

Auto Correct option facilitates the user in such a fashion that it automatically corrects common typing or spelling errors as it types.



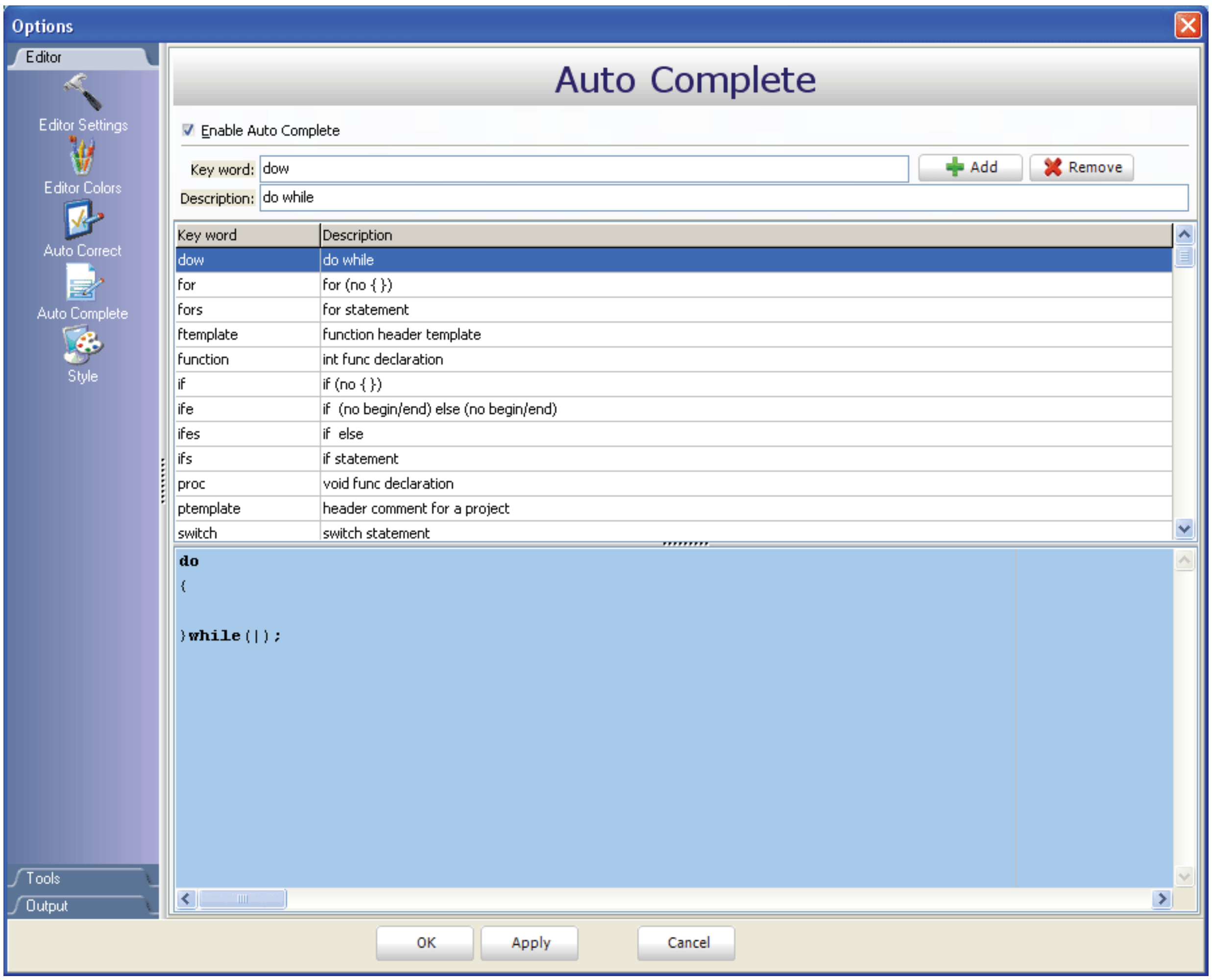
This option is already set up to automatically correct some words. For example, if you type `whiel`, it will be corrected to `while` when you press the spacebar:



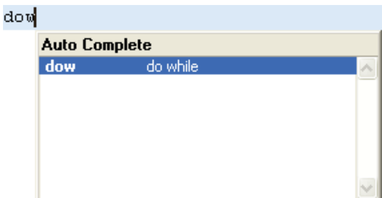
The user can easily add its common typos by entering original typo, for example `btye`, to the Original box, and replacement, `byte`, to the Replacement box, and just click "Add" button.
Next time when the typo occurs, it will be automatically corrected.

Auto Complete (Code Templates)

Auto Complete option saves lots of keystrokes for commonly used phrases by automatically completing user's typing.



The user can insert the Code Template by typing the name of the template (for instance, `dow`), then press Ctrl+J and the Code Editor will automatically generate a code:



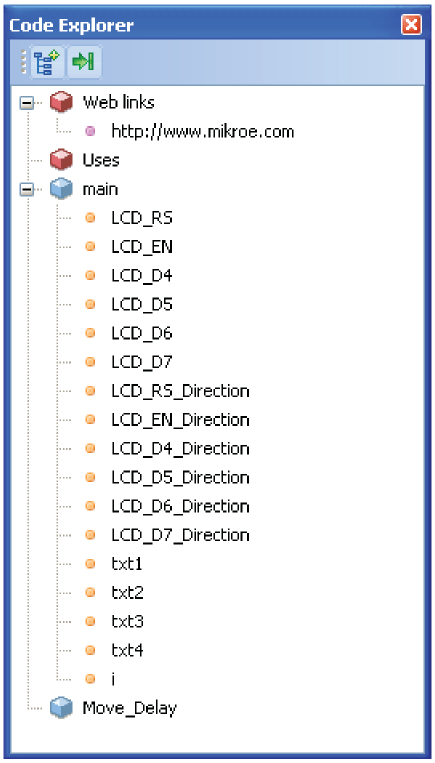
You can add your own templates to the list by entering the desired keyword, description and code of your template in appropriate boxes.
Autocomplete macros can retreive system and project information:

- `%DATE%` - current system date
- `%TIME%` - current system time
- `%DEVICE%` - device (MCU) name as specified in project settings
- `%DEVICE_CLOCK%` - clock as specified in project settings
- `%COMPILER%` - current compiler version

These macros can be used in template code, see template `ptemplate` provided with mikroPascal PRO for PIC32 installation.

Code Explorer

The Code Explorer gives clear view of each item declared inside the source code. You can jump to a declaration of any item by double clicking it, or pressing the Enter button. Also, besides the list of defined and declared objects, code explorer displays message about the first error and it's location in code.



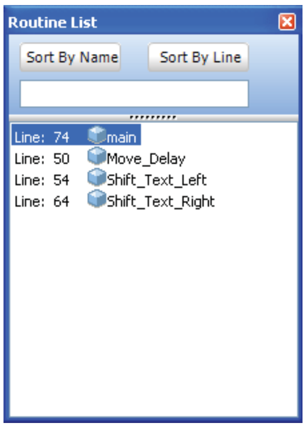
The following options are available in the Code Explorer:

Icon	Description
	Expand/Collapse all nodes in tree.
	Locate declaration in code.

Routine List

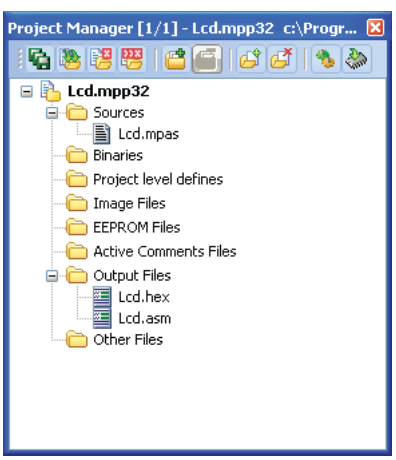
Routine list diplays list of routines, and enables filtering routines by name. Routine list window can be accessed by pressing Ctrl+L.

You can jump to a desired routine by double clicking on it, or pressing the Enter button. Also, you can sort routines by size or by address.



Project Manager

Project Manager is IDE feature which allows the users to manage multiple projects. Several projects which together make project group may be open at the same time. Only one of them may be active at the moment.
Setting project in **active** mode is performed by **double clicking** the desired project in the Project Manager, which will result in bolding the project's name.
Also, the name of the currently active project will be diplayed in the Program Manager window title, alongside with the number of projects in project group.



Following options are available in the Project Manager:

Icon	Description
	Save project Group.
	Open project group.
	Close the active project.
	Close project group.
	Add project to the project group.
	Remove project from the project group.
	Add file to the active project.
	Remove selected file from the project.
	Build the active project.
	Run mikroElektronika's Flash programmer.

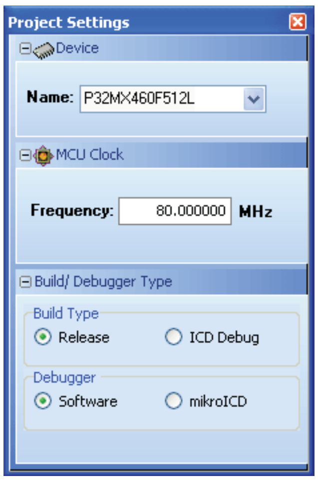
For details about adding and removing files from project see Add/Remove Files from Project.

Related topics: Project Settings, Project Menu Options, File Menu Options, Project Toolbar, Build Toolbar, Add/Remove Files from Project

Project Settings

The following options are available in the Project Settings window:

- Device - select the appropriate device from the device drop-down list.
- MCU Clock - enter the clock frequency value.
- Build/Debugger Type - choose debugger type.



Related topics: Edit Project, Customizing Projects, Project Manager

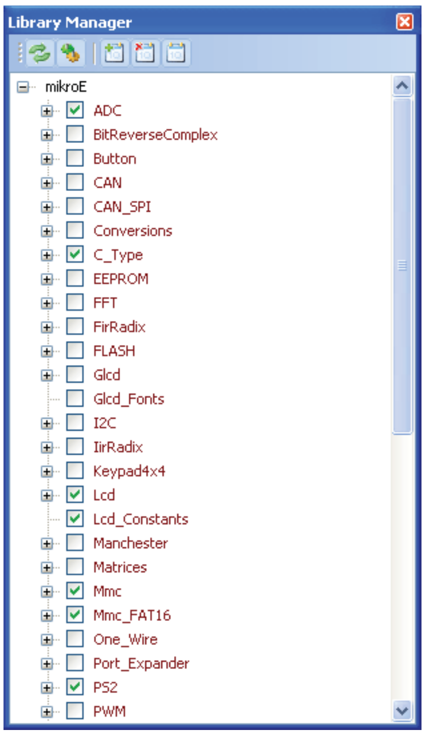
Library Manager

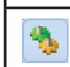
Library Manager enables simple handling libraries being used in a project. Library Manager window lists all libraries (extension `.emcl`) which are instantly stored in the compiler *Uses* folder. The desirable library is added to the project by selecting check box next to the library name.

In order to have all library functions accessible, simply press the button **Check All**  and all libraries will be selected.

In case none library is needed in a project, press the button **Clear All**  and all libraries will be cleared from the project.

Only the selected libraries will be linked.

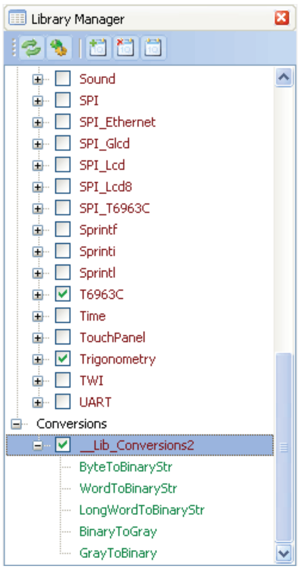


Icon	Description
	Refresh Library by scanning files in “Uses” folder. Useful when new libraries are added by copying files to “Uses” folder.
	Rebuild all available libraries. Useful when library sources are available and need refreshing.
	Include all available libraries in current project.
	No libraries from the list will be included in current project.
	Restore library to the state just before last project saving.

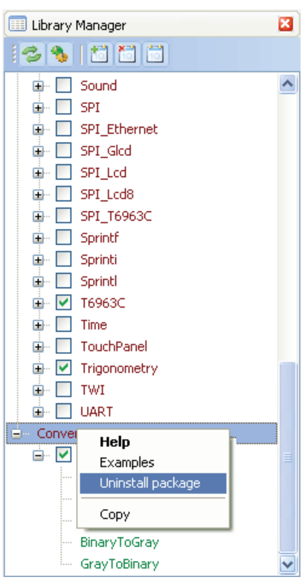
Managing libraries using Package Manager

The Package Manager is a tool which enables users to easily install their own libraries in the mikroIDE. Libraries are distributed in the form of a package, which is an archive composed of one or more files, containing libraries. For more information on Package Manager, visit our website.

Upon package installation, a new node with the package name will be created in the Library Manager. For example:



From the Library Manager, the user can also uninstall the desired package by right clicking the the appropriate node, and from the drop-down menu choose Uninstall package:

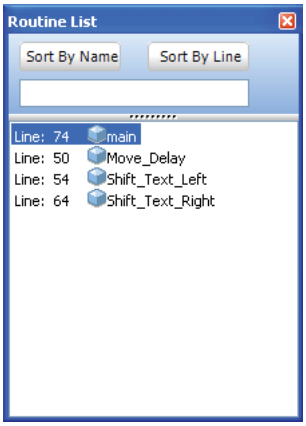


Related topics: mikroPascal PRO for PIC Libraries, Creating New Library

Routine List

Routine list diplays list of routines, and enables filtering routines by name. Routine list window can be accessed by pressing Ctrl+L.

You can jump to a desired routine by double clicking on it, or pressing the Enter button. Also, you can sort routines by size or by address.



Statistics

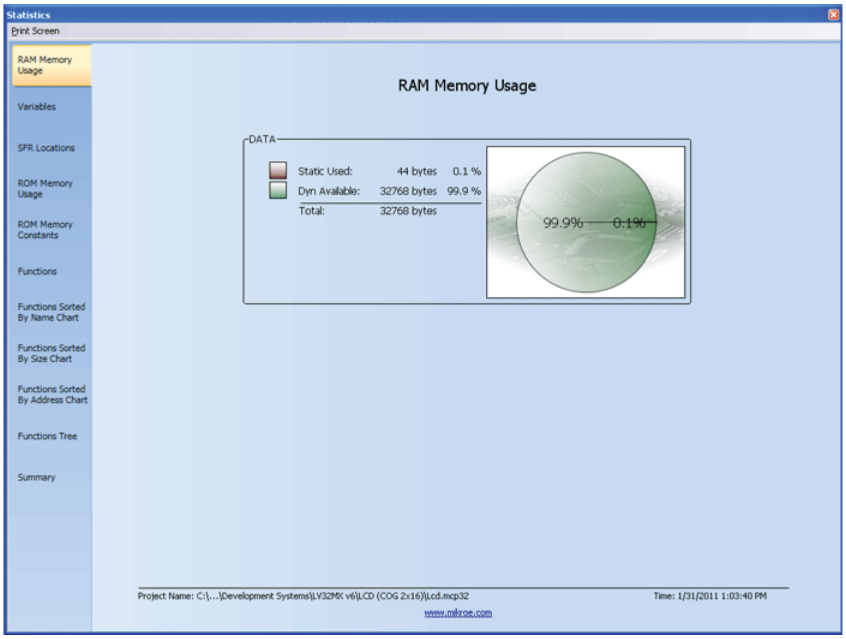
After successful compilation, you can review statistics of your code. Click the Statistics Icon  .

Memory Usage Windows

Provides overview of RAM and ROM usage in the various forms.

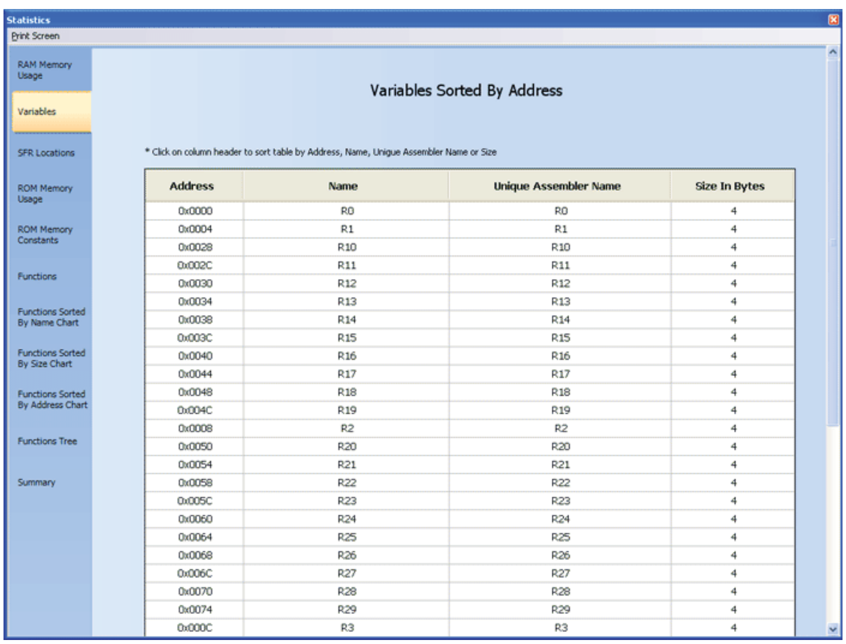
Variables

Displays variables sorted by addresses.



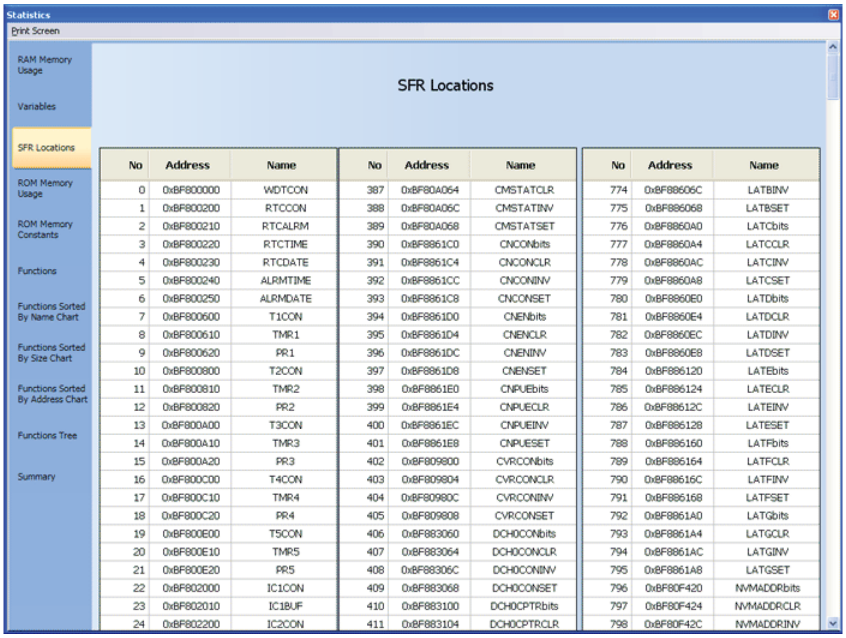
Used RAM Locations

Displays used RAM memory locations and their names.



SFR Locations

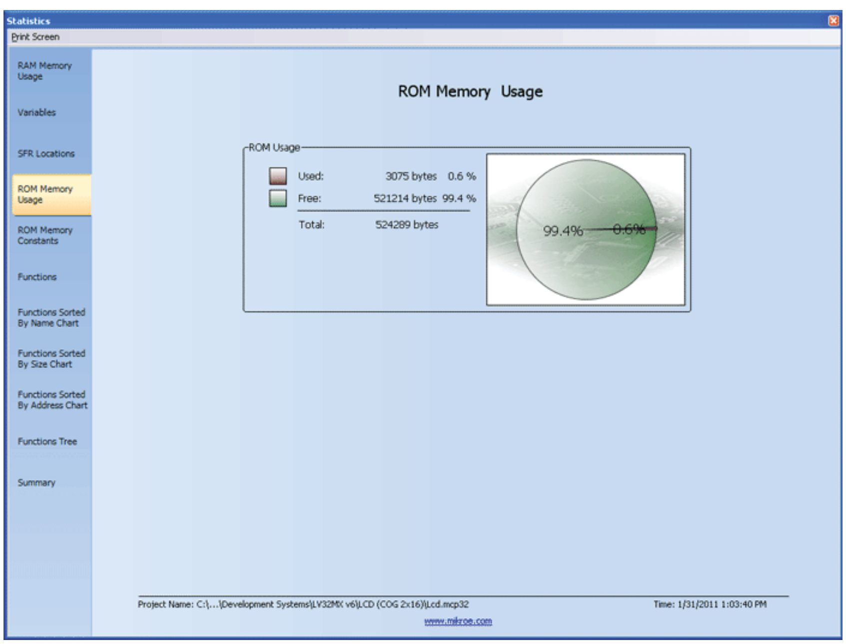
Displays list of used SFR locations.



No	Address	Name	No	Address	Name	No	Address	Name
0	0xBF800000	WDTCON	387	0xBF80A064	CMSTATCLR	774	0xBF88606C	LATBINV
1	0xBF800200	RTCCON	388	0xBF80A06C	CMSTATINV	775	0xBF886068	LATBSET
2	0xBF800210	RTCALRM	389	0xBF80A068	CMSTATSET	776	0xBF8860A0	LATCbits
3	0xBF800220	RTCTIME	390	0xBF8861C0	CNCONbits	777	0xBF8860A4	LATCCLR
4	0xBF800230	RTCDATE	391	0xBF8861C4	CNCONCLR	778	0xBF8860AC	LATCINV
5	0xBF800240	ALRMTIME	392	0xBF8861CC	CNCONINV	779	0xBF8860A8	LATCSET
6	0xBF800250	ALRMDATE	393	0xBF8861C8	CNCONSET	780	0xBF8860E0	LATDbits
7	0xBF800600	T1CON	394	0xBF8861D0	CNENbits	781	0xBF8860E4	LATDCLR
8	0xBF800610	TMR1	395	0xBF8861D4	CNENCLR	782	0xBF8860EC	LATDINV
9	0xBF800620	PR1	396	0xBF8861DC	CNENINV	783	0xBF8860E8	LATDSET
10	0xBF800800	T2CON	397	0xBF8861D8	CNENSET	784	0xBF886120	LATEbits
11	0xBF800810	TMR2	398	0xBF8861E0	CNPUEbits	785	0xBF886124	LATECLR
12	0xBF800820	PR2	399	0xBF8861E4	CNPUECLR	786	0xBF88612C	LATEINV
13	0xBF800A00	T3CON	400	0xBF8861EC	CNPUEINV	787	0xBF886128	LATESET
14	0xBF800A10	TMR3	401	0xBF8861E8	CNPUESET	788	0xBF886160	LATFbits
15	0xBF800A20	PR3	402	0xBF809800	CVRCONbits	789	0xBF886164	LATFCLR
16	0xBF800C00	T4CON	403	0xBF809804	CVRCONCLR	790	0xBF88616C	LATFINV
17	0xBF800C10	TMR4	404	0xBF80980C	CVRCONINV	791	0xBF886168	LATFSET
18	0xBF800C20	PR4	405	0xBF809808	CVRCONSET	792	0xBF8861A0	LATGbits
19	0xBF800E00	T5CON	406	0xBF883060	DCH0CONbits	793	0xBF8861A4	LATGCLR
20	0xBF800E10	TMR5	407	0xBF883064	DCH0CONCLR	794	0xBF8861AC	LATGINV
21	0xBF800E20	PR5	408	0xBF88306C	DCH0CONINV	795	0xBF8861A8	LATGSET
22	0xBF802000	IC1CON	409	0xBF883068	DCH0CONSET	796	0xBF80F420	NVMADDRbits
23	0xBF802010	IC1BUF	410	0xBF883100	DCH0CPTRbits	797	0xBF80F424	NVMADDRCLR
24	0xBF802200	IC2CON	411	0xBF883104	DCH0CPTRCLR	798	0xBF80F42C	NVMADDRINV

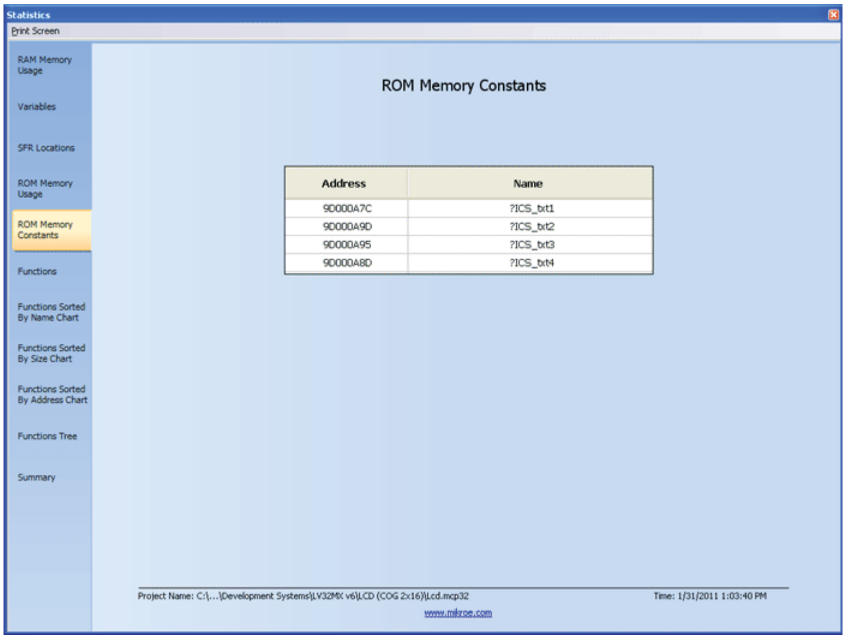
ROM Memory Usage

Displays ROM memory space usage in a pie-like form.



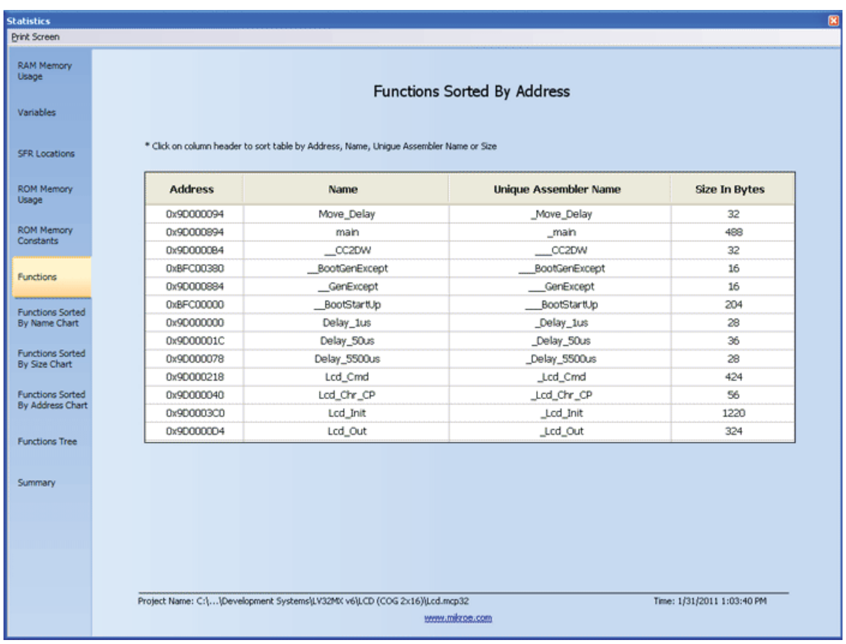
ROM Memory Constants

Displays ROM memory constants and their addresses.



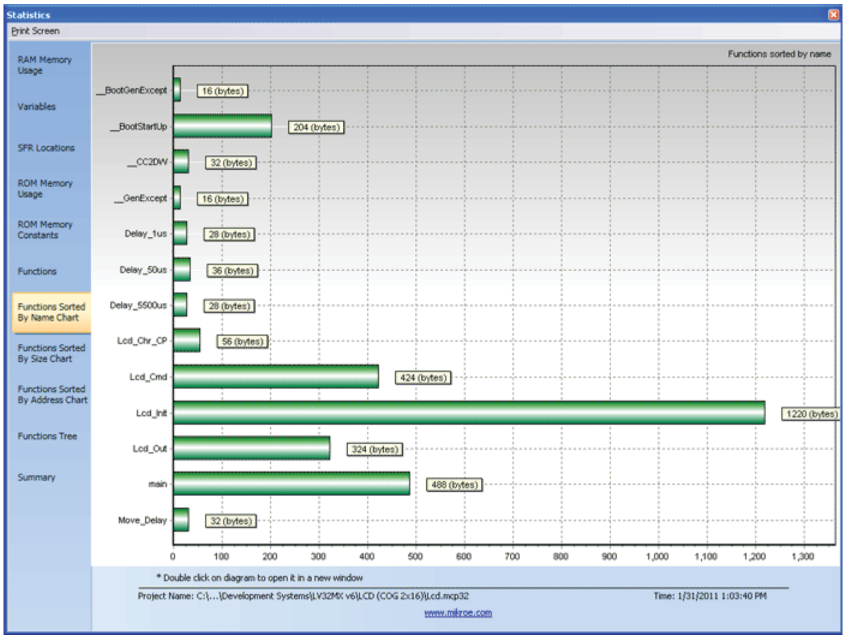
Functions

Sorts and displays functions in various ways.



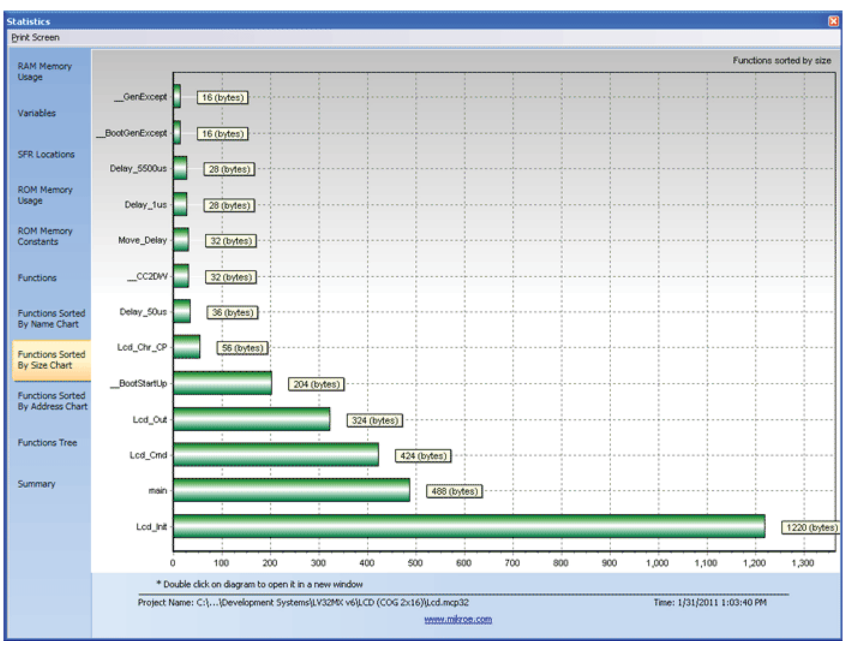
Functions Sorted By Name Chart

Sorts and displays functions by their name, in the ascending order.



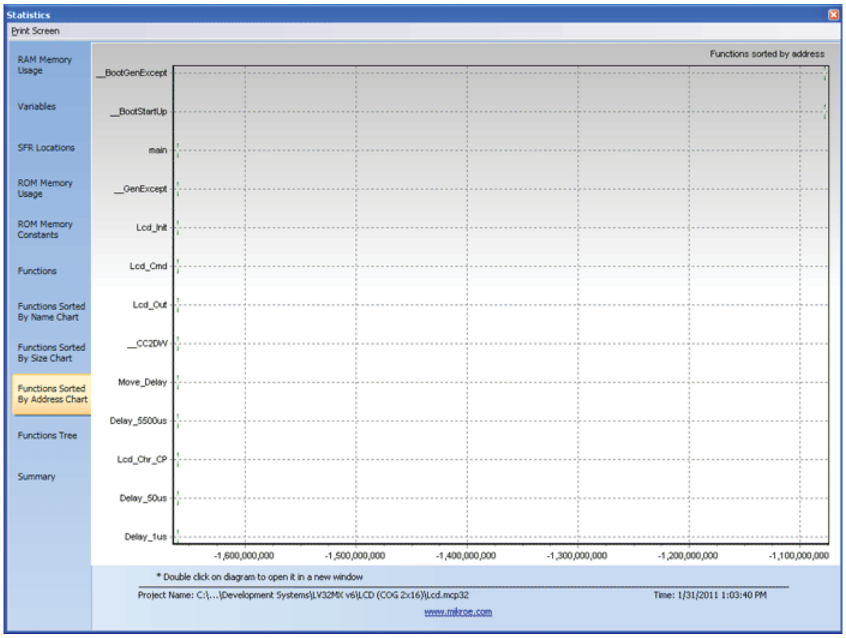
Functions Sorted By Size Chart

Sorts and displays functions by their sizes in a chart-like form.



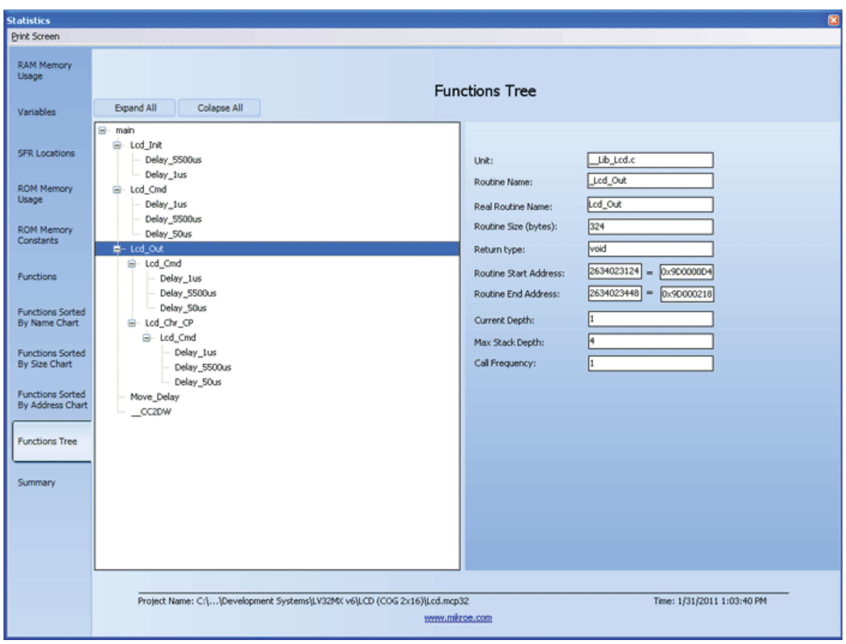
Functions Sorted By Addresses

Sorts and displays functions by their addresses, in the ascending order.



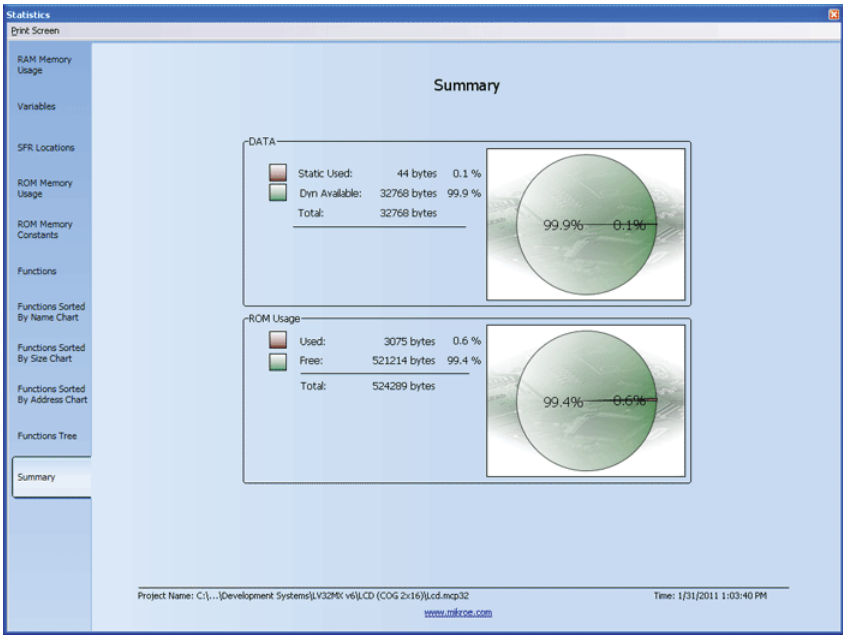
Function Tree

Displays Function Tree with the relevant data for each function.



Memory Summary

Displays summary of RAM and ROM memory in a pie-like form.

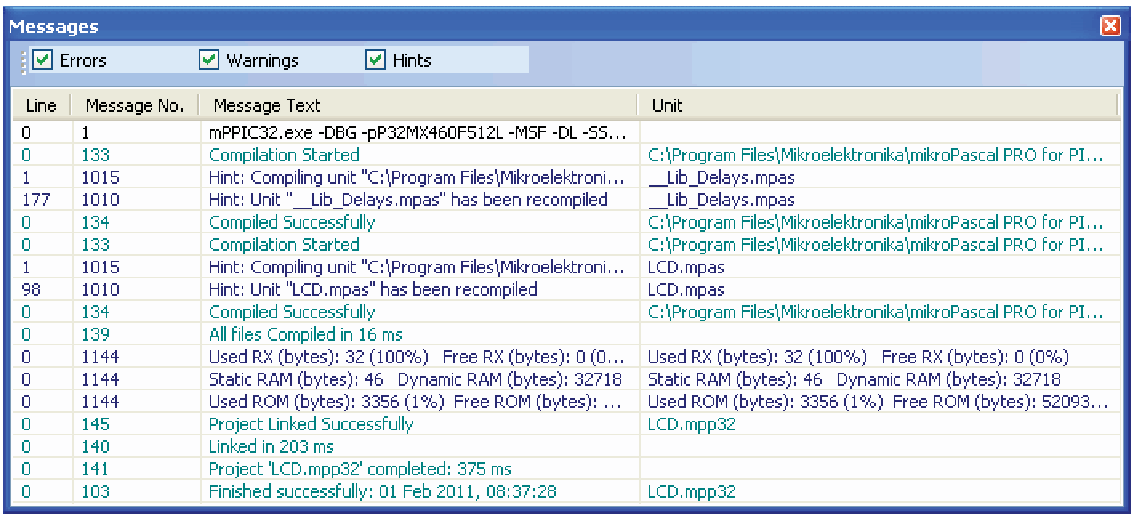


Messages Window

Messages Window displays various informations and notifications about the compilation process.

It reports for example, time needed for preprocessing, compilation and linking; used RAM and ROM space, generated baud rate with error percentage, etc.
The user can filter which notifications will Messages Window display by checking Errors, Warning and Hints box.

In case that errors were encountered during compiling, the compiler will report them and won’t generate a hex file. The Messages Window will display errros at the bottom of the window by default.
The compiler also reports warnings, but these do not affect the output; only errors can interefere with the generation of hex.

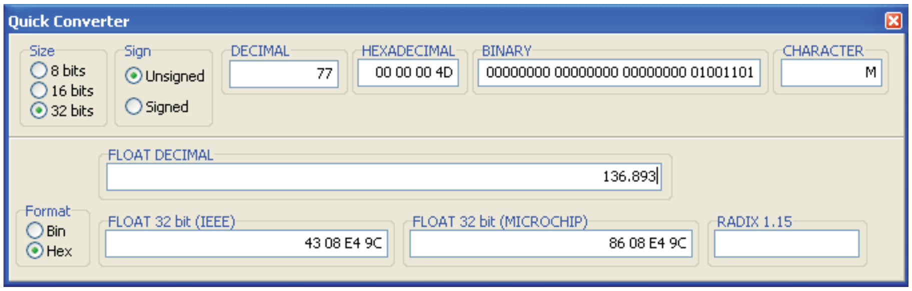


Line	Message No.	Message Text	Unit
0	1	mPPIC32.exe -DBG -pP32MX460F512L -MSF -DL -SS...	
0	133	Compilation Started	C:\Program Files\Mikroelektronika\mikroPascal PRO for PI...
1	1015	Hint: Compiling unit "C:\Program Files\Mikroelektroni...	__Lib_Delays.mpas
177	1010	Hint: Unit "__Lib_Delays.mpas" has been recompiled	__Lib_Delays.mpas
0	134	Compiled Successfully	C:\Program Files\Mikroelektronika\mikroPascal PRO for PI...
0	133	Compilation Started	C:\Program Files\Mikroelektronika\mikroPascal PRO for PI...
1	1015	Hint: Compiling unit "C:\Program Files\Mikroelektroni...	LCD.mpas
98	1010	Hint: Unit "LCD.mpas" has been recompiled	LCD.mpas
0	134	Compiled Successfully	C:\Program Files\Mikroelektronika\mikroPascal PRO for PI...
0	139	All files Compiled in 16 ms	
0	1144	Used RX (bytes): 32 (100%) Free RX (bytes): 0 (0...	Used RX (bytes): 32 (100%) Free RX (bytes): 0 (0%)
0	1144	Static RAM (bytes): 46 Dynamic RAM (bytes): 32718	Static RAM (bytes): 46 Dynamic RAM (bytes): 32718
0	1144	Used ROM (bytes): 3356 (1%) Free ROM (bytes): ...	Used ROM (bytes): 3356 (1%) Free ROM (bytes): 52093...
0	145	Project Linked Successfully	LCD.mpp32
0	140	Linked in 203 ms	
0	141	Project 'LCD.mpp32' completed: 375 ms	
0	103	Finished successfully: 01 Feb 2011, 08:37:28	LCD.mpp32

Double click the message line in the Message Window to highlight the line where the error was encountered.

Quick Converter

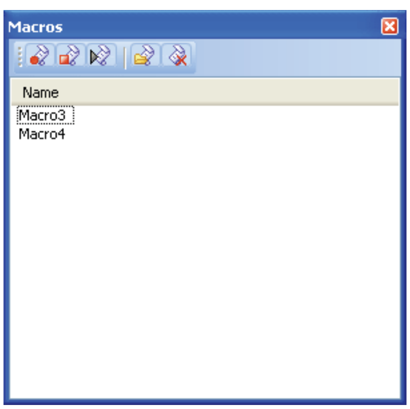
Quick Converter enables the user to easily transform numbers from one base to another.



The user can convert integers of various sizes (8, 16 or 32 bits), signed and unsigned, using different representation (decimal, hexadecimal, binary and character).
Also, Quick Converter features float point numbers conversion from/to Float Decimal, Float 32bit (IEEE), Float 32bit (Microchip) and Radix 1.15 for PIC32 family of MCUs.

Macro Editor

A macro is a series of keystrokes that have been 'recorded' in the order performed. A macro allows you to 'record' a series of keystrokes and then 'playback', or repeat, the recorded keystrokes.



The Macro offers the following commands:

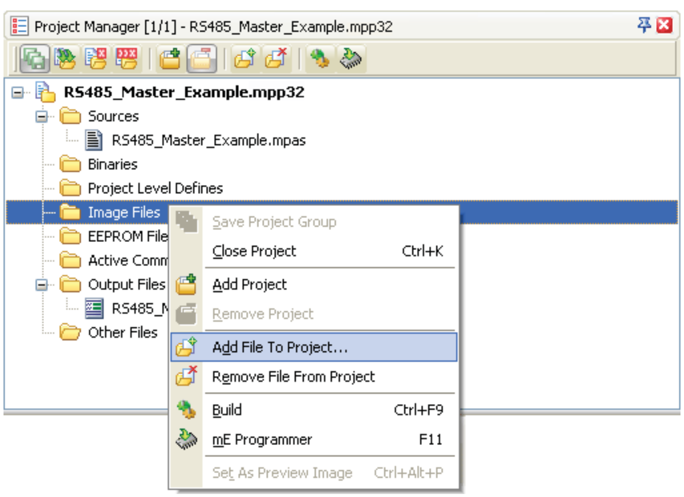
Icon	Description
	Starts ‘recording’ keystrokes for later playback.
	Stops capturing keystrokes that was started when the Start Recording command was selected.
	Allows a macro that has been recorded to be replayed.
	New macro.
	Delete macro.

Related topics: Code Editor, Code Templates

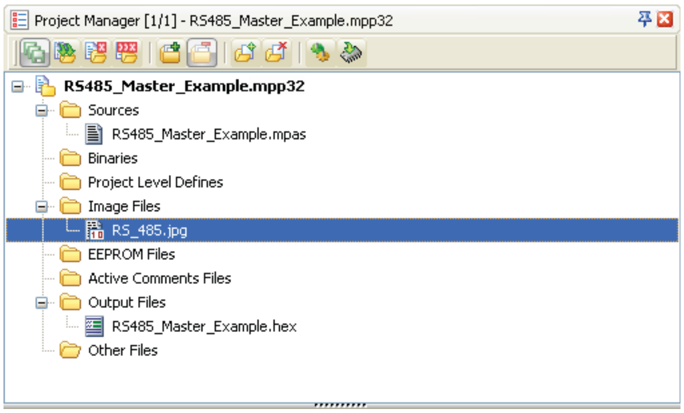
Image Preview

There are a lot of occassions in which the user besides the code, must look at the appropriate schematics in order to succesfully write the desired program.
The mikroPascal PRO for PIC32 provides this possibility through the **Image Preview Window**.

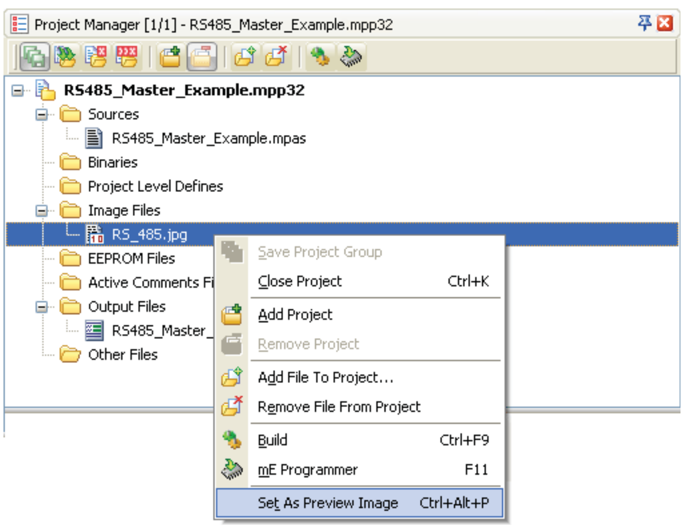
To add an image to the **Image Preview Window**, right click the **Image Files** node in the **Project Manager**:



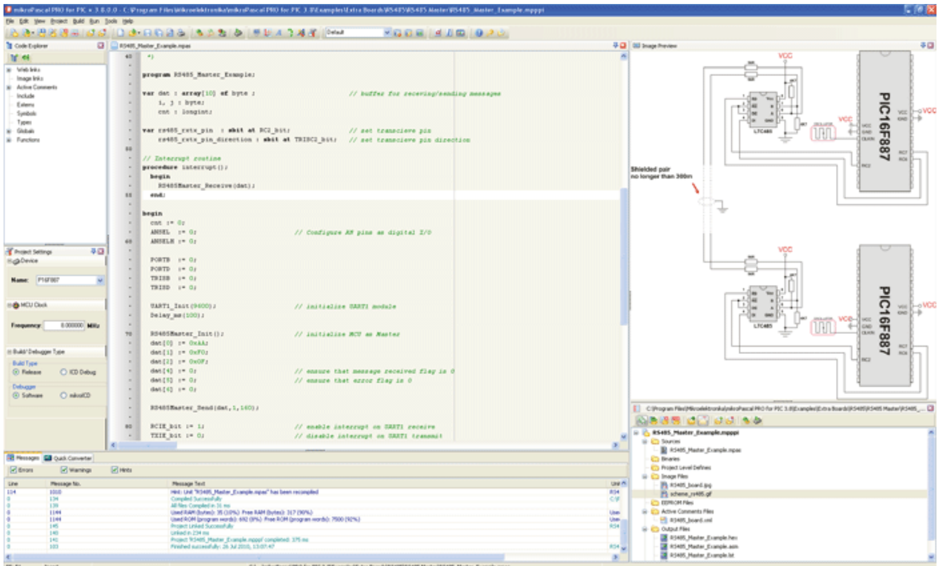
Now, navigate to the desired image file, and simply add it:



Next, right click the added file, and choose **Set As Preview Image**:



Once you have added the image, it will appear in the **Image Preview Window**:



Also, you can add multiple images to the **Image Files** node, but only the one that is set will be automatically displayed in the **Image Preview Window** upon opening the project.

By changing the **Image Preview Window** size, displayed image will be fit by its height in such a way that its proportions will remain intact.

Toolbars

This section provides an overview of the toolbars available in mikroPascal PRO for PIC32 Help:

- File Toolbar
- Edit Toolbar
- Advanced Edit Toolbar
- Find Toolbar
- Project Toolbar
- Build Toolbar
- Debug Toolbar
- Styles Toolbar
- Tools Toolbar
- View Toolbar
- Layout Toolbar
- Help Toolbar

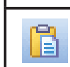
File Toolbar

File Toolbar is a standard toolbar with the following options:

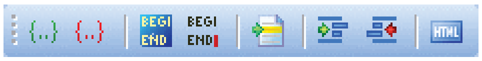
Icon	Description
	Opens a new editor window.
	Open source file for editing or image file for viewing.
	Save changes for active window.
	Save changes in all opened windows.
	Print Preview.
	Print.

Edit Toolbar

Edit Toolbar is a standard toolbar with the following options:

Icon	Description
	Undo last change.
	Redo last change.
	Cut selected text to clipboard.
	Copy selected text to clipboard.
	Paste text from clipboard.

Advanced Edit Toolbar



Advanced Edit Toolbar comes with the following options:

Icon	Description
	Comment selected code or put a single line comment if there is no selection
	Uncomment selected code or remove single line comment if there is no selection.
	Select text from starting delimiter to ending delimiter.
	Go to ending delimiter.
	Go to line.
	Indent selected code lines.
	Outdent selected code lines.
	Generate HTML code suitable for publishing current source code on the web.

Find/Replace Toolbar

Find/Replace Toolbar is a standard toolbar with the following options:

Icon	Description
	Find text in current editor.
	Find next occurence.
	Find previous occurence.
	Replace text.
	Find text in files.

Project Toolbar

Project Toolbar comes with the following options:

Icon	Description
	New project.
	Open Project
	Save Project
	Edit project settings.
	Close current project.
	Clean project folder.
	Add File To Project
	Remove File From Project

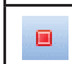
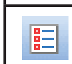
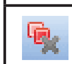
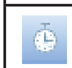
Build Toolbar

Build Toolbar comes with the following options:

Icon	Description
	Build current project.
	Build all opened projects.
	Build and program active project.
	Start programmer and load current HEX file.

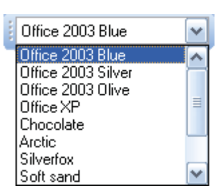
Debug Toolbar

Debug Toolbar comes with the following options:

Icon	Description
	Start Software Simulator or mikroICD (In-Circuit Debugger).
	Run/Pause Debugger.
	Stop Debugger.
	Step Into.
	Step Over.
	Step Out.
	Run To Cursor.
	Toggle Breakpoint.
	View Breakpoints Window
	Clear Breakpoints.
	View Watch Window
	View Stopwatch Window

Styles Toolbar

Styles toolbar allows you to easily change colors of your workspace.



Tools Toolbar

Tools Toolbar comes with the following default options:

Icon	Description
	Run USART Terminal
	EEPROM
	ASCII Chart
	Seven Segment Editor.
	Open Active Comment editor.
	Options menu

Tip : The Tools toolbar can easily be customized by adding new tools in Options menu window.

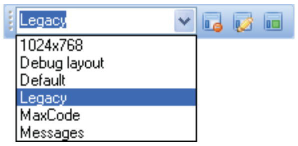
View Toolbar

View Toolbar provides access to assembly code, listing file and statistics windows.

Icon	Description
	Open assembly code in editor.
	Open listing file in editor.
	View statistics for current project.

Layout Toolbar

Styles toolbar allows you to easily customize workspace through a number of different IDE layouts.



Help Toolbar

Help Toolbar provides access to information on using and registering compilers:

Icon	Description
	Open Help file.
	How To Register.

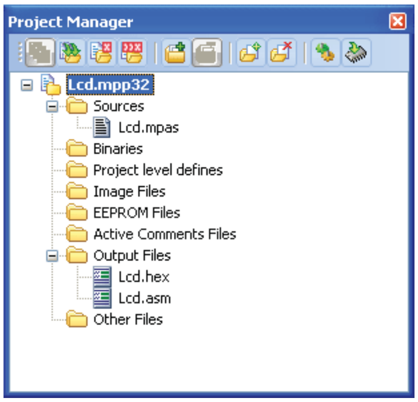
Related topics: Keyboard shortcuts, Integrated Tools

Customizing IDE Layout

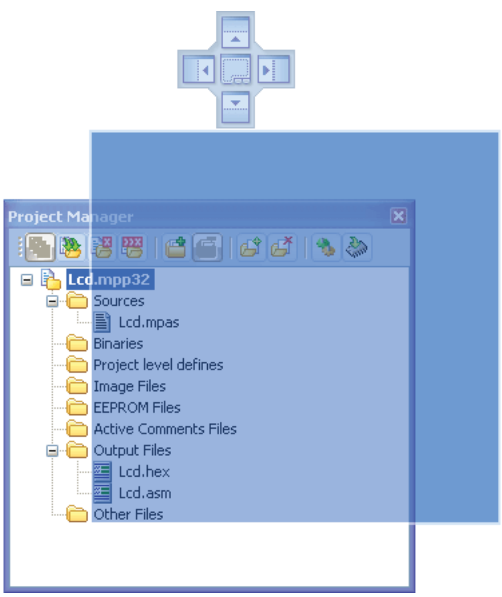
Docking Windows

You can increase the viewing and editing space for code, depending on how you arrange the windows in the IDE.

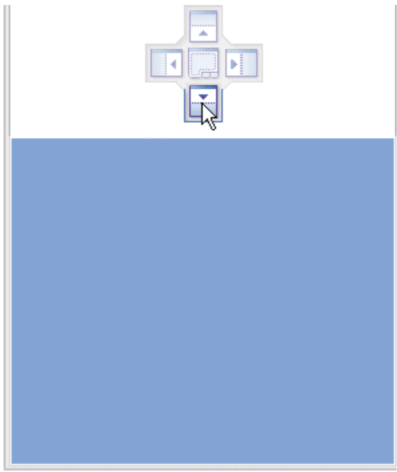
Step 1: Click the window you want to dock, to give it focus.



Step 2: Drag the tool window from its current location. A guide diamond appears. The four arrows of the diamond point towards the four edges of the IDE.



Step 3: Move the pointer over the corresponding portion of the guide diamond. An outline of the window appears in the designated area.



Step 4: To dock the window in the position indicated, release the mouse button.

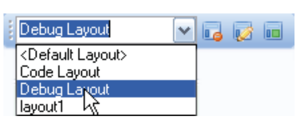
Tip : To move a dockable window without snapping it into place, press CTRL while dragging it.

Saving Layout

Once you have a window layout that you like, you can save the layout by typing the name for the layout and pressing the Save Layout Icon  .

To set the layout select the desired layout from the layout drop-down list and click the Set Layout Icon  .

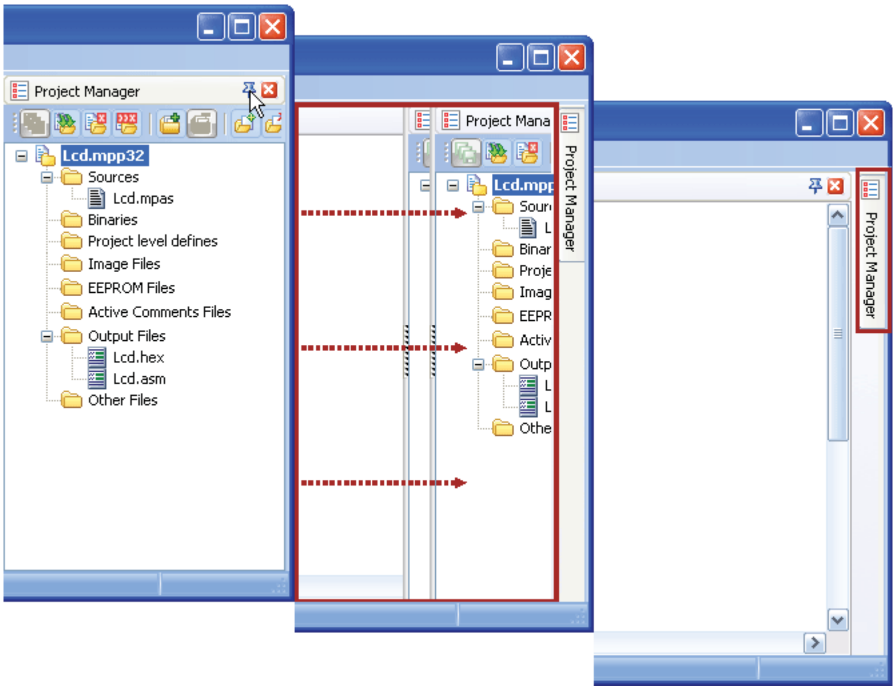
To remove the layout from the drop-down list, select the desired layout from the list and click the Delete Layout Icon  .



Auto Hide

Auto Hide enables you to see more of your code at one time by minimizing tool windows along the edges of the IDE when not in use.

- Click the window you want to keep visible to give it focus.
- Click the Pushpin Icon  on the title bar of the window.



When an auto-hidden window loses focus, it automatically slides back to its tab on the edge of the IDE. While a window is auto-hidden, its name and icon are visible on a tab at the edge of the IDE. To display an auto-hidden window, move your pointer over the tab. The window slides back into view and is ready for use.

Options

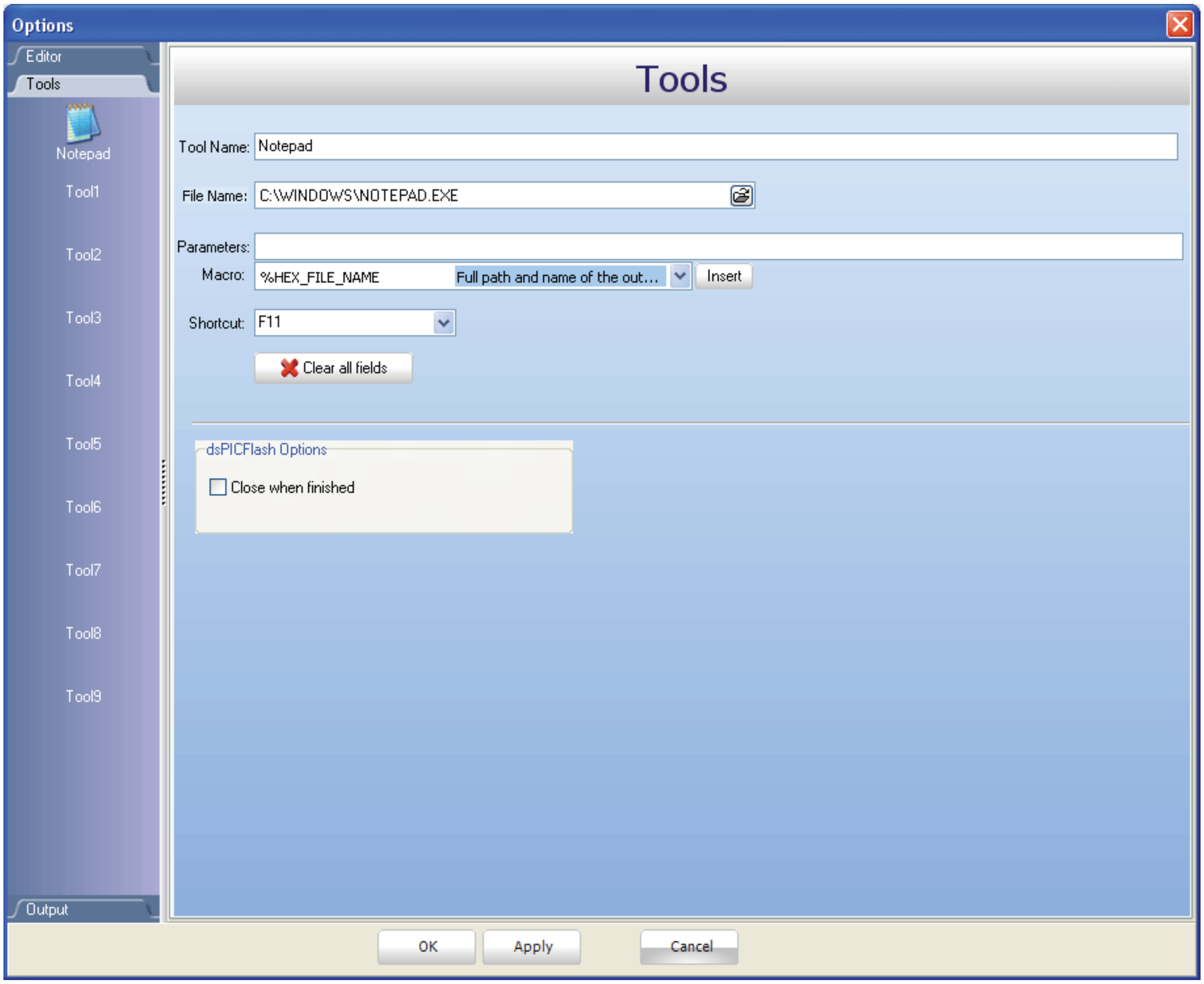
Options menu consists of three tabs: Code Editor, Tools and Output settings.

Code editor

The Code Editor is advanced text editor fashioned to satisfy needs of professionals.

Tools

The mikroPascal PRO for PIC32 includes the Tools tab, which enables the use of shortcuts to external programs, like Calculator or Notepad.
You can set up to 10 different shortcuts, by editing Tool0 - Tool9.



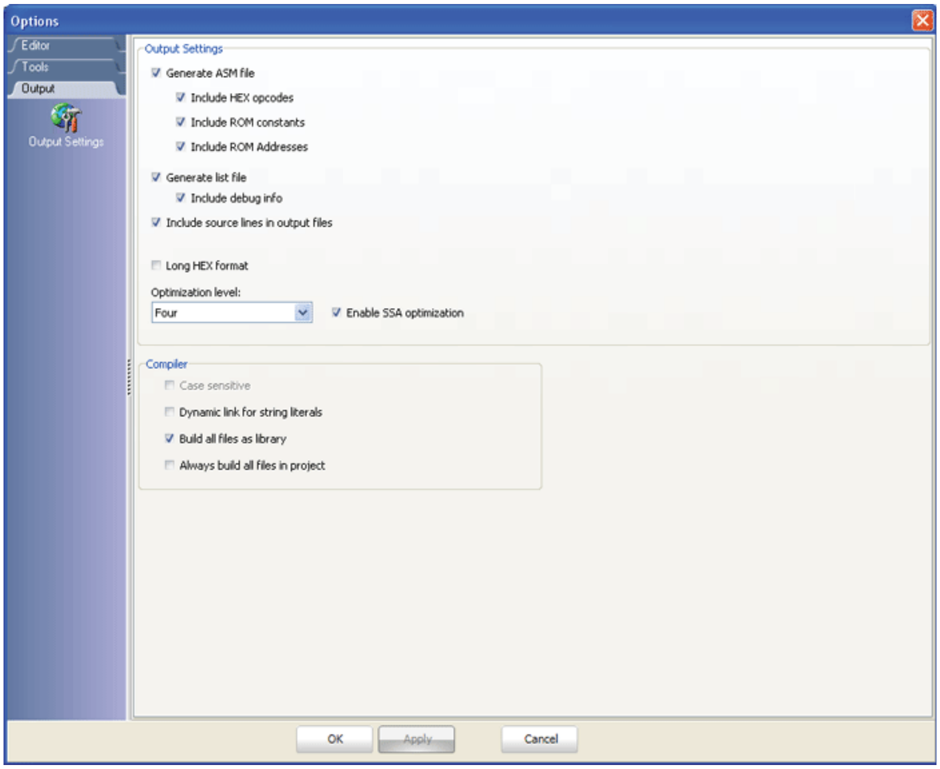
Output settings

By modifying Output Settings, user can configure the content of the output files.
You can enable or disable, for example, generation of ASM and List file.

Also, user can choose optimization level, and compiler specific settings, which include case sensitivity, dynamic link for string literals setting (described in mikroPascal PRO for PIC32 specifics).

Build all files as library enables user to use compiled library (`*.emcl`) on any MCU (when this box is checked), or for a selected MCU (when this box is left unchecked).

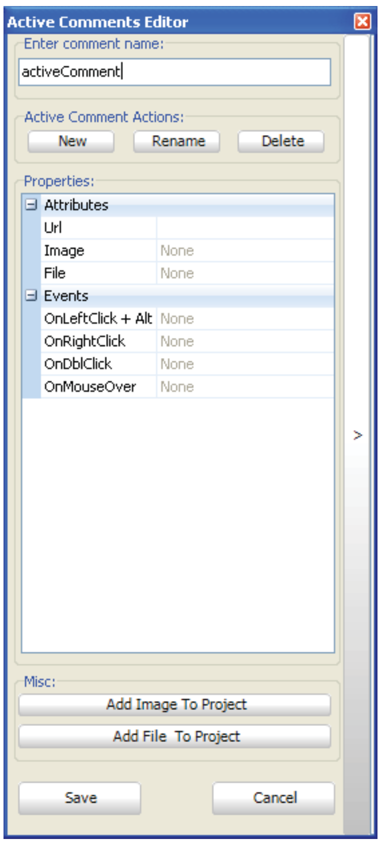
For more information on creating new libraries, see Creating New Library.


Options
Editor
Tools
Output
Output Settings
Output Settings
Generate ASM file
Include HEX opcodes
Include ROM constants
Include ROM Addresses
Generate list file
Include debug info
Include source lines in output files
Long HEX format
Optimization level:
Four
Enable SSA optimization
Compiler
Case sensitive
Dynamic link for string literals
Build all files as library
Always build all files in project
OK
Apply
Cancel

Integrated Tools

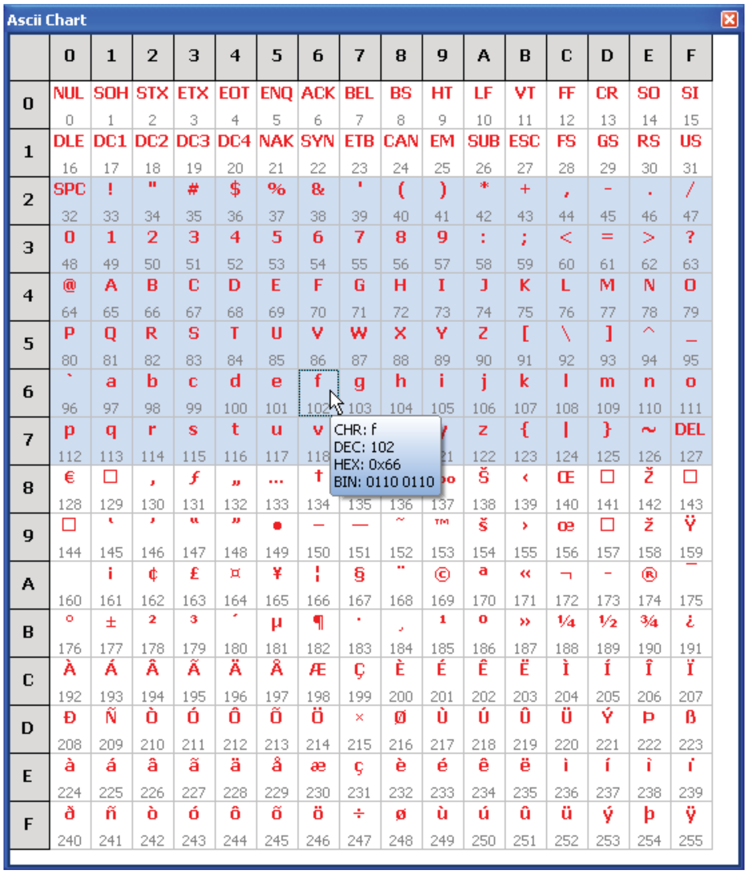
Active Comments Editor

Active Comments Editor is a tool, particularly useful when working with Lcd display. You can launch it from the drop-down menu **Tools › Active Comments Editor** or by clicking the Active Comment Editor Icon from Tools toolbar.



ASCII Chart

The ASCII Chart is a handy tool, particularly useful when working with Lcd display. You can launch it from the drop-down menu **Tools › ASCII chart** or by clicking the View ASCII Chart Icon from Tools toolbar.



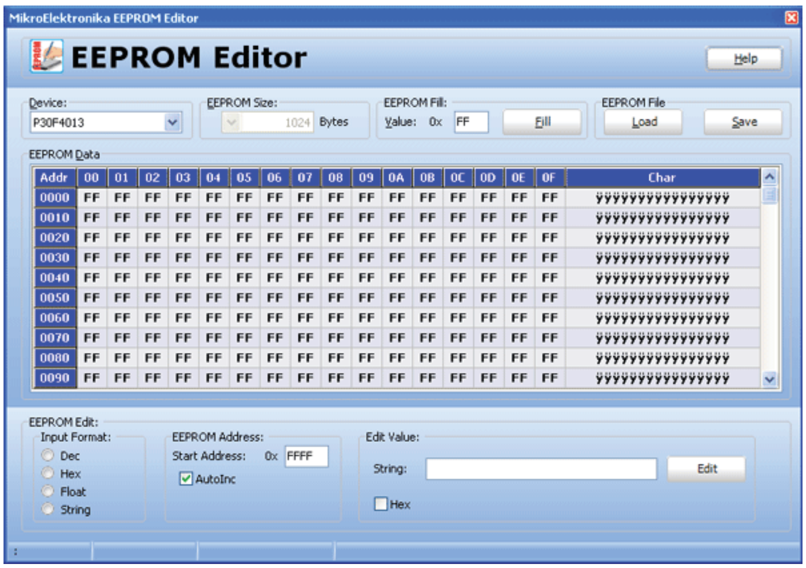
Ascii Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL 0	SOH 1	STX 2	ETX 3	EOT 4	ENQ 5	ACK 6	BEL 7	BS 8	HT 9	LF 10	VT 11	FF 12	CR 13	SO 14	SI 15
1	DLE 16	DC1 17	DC2 18	DC3 19	DC4 20	NAK 21	SYN 22	ETB 23	CAN 24	EM 25	SUB 26	ESC 27	FS 28	GS 29	RS 30	US 31
2	SPC 32	! 33	" 34	# 35	$ 36	% 37	& 38	' 39	(40	) 41	* 42	+ 43	, 44	- 45	. 46	/ 47
3	0 48	1 49	2 50	3 51	4 52	5 53	6 54	7 55	8 56	9 57	: 58	; 59	< 60	= 61	> 62	? 63
4	@ 64	A 65	B 66	C 67	D 68	E 69	F 70	G 71	H 72	I 73	J 74	K 75	L 76	M 77	N 78	O 79
5	P 80	Q 81	R 82	S 83	T 84	U 85	V 86	W 87	X 88	Y 89	Z 90	[91	\ 92	] 93	^ 94	_ 95
6	` 96	a 97	b 98	c 99	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
7	p 112	q 113	r 114	s 115	t 116	u 117	v 118	[illegible]	[illegible]	[illegible]	z 122	{ 123	| 124	} 125	~ 126	DEL 127
8	€ 128	□ 129	‚ 130	ƒ 131	„ 132	… 133	† 134	[illegible] 135	[illegible] 136	[illegible] 137	Š 138	‹ 139	Œ 140	□ 141	Ž 142	□ 143
9	□ 144	‘ 145	’ 146	“ 147	” 148	• 149	– 150	— 151	˜ 152	™ 153	š 154	› 155	œ 156	□ 157	ž 158	Ÿ 159
A	160	¡ 161	¢ 162	£ 163	¤ 164	¥ 165	¦ 166	§ 167	¨ 168	© 169	ª 170	« 171	¬ 172	- 173	® 174	¯ 175
B	° 176	± 177	² 178	³ 179	´ 180	µ 181	¶ 182	· 183	¸ 184	¹ 185	º 186	» 187	¼ 188	½ 189	¾ 190	¿ 191
C	À 192	Á 193	Â 194	Ã 195	Ä 196	Å 197	Æ 198	Ç 199	È 200	É 201	Ê 202	Ë 203	Ì 204	Í 205	Î 206	Ï 207
D	Ð 208	Ñ 209	Ò 210	Ó 211	Ô 212	Õ 213	Ö 214	× 215	Ø 216	Ù 217	Ú 218	Û 219	Ü 220	Ý 221	Þ 222	ß 223
E	à 224	á 225	â 226	ã 227	ä 228	å 229	æ 230	ç 231	è 232	é 233	ê 234	ë 235	ì 236	í 237	î 238	ï 239
F	ð 240	ñ 241	ò 242	ó 243	ô 244	õ 245	ö 246	÷ 247	ø 248	ù 249	ú 250	û 251	ü 252	ý 253	þ 254	ÿ 255

EEPROM Editor

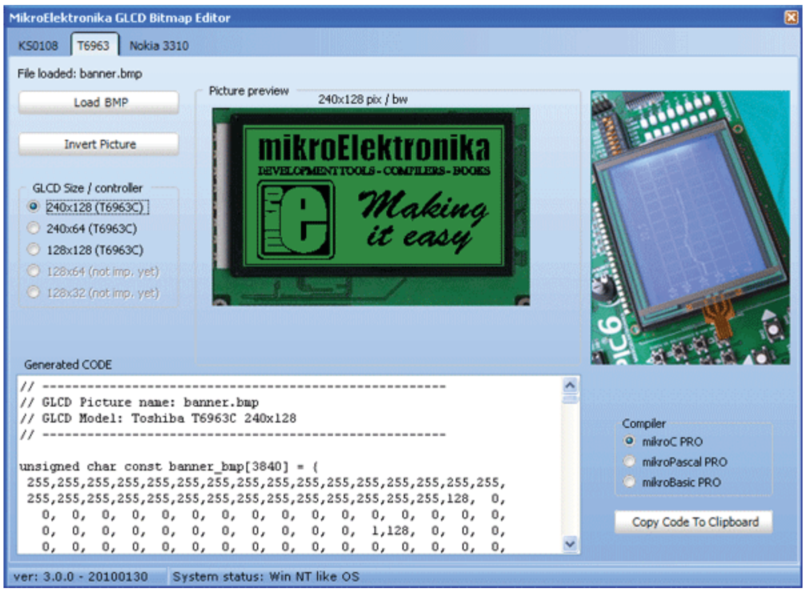
The EEPROM Editor is used for manipulating MCU's EEPROM memory. You can launch it from the drop-down menu **Tools › EEPROM Editor**.

When you run mikroElektronika programmer software from mikroPascal PRO for PIC32 IDE - `project_name.hex` file will be loaded automatically while `ihex` file must be loaded manually.



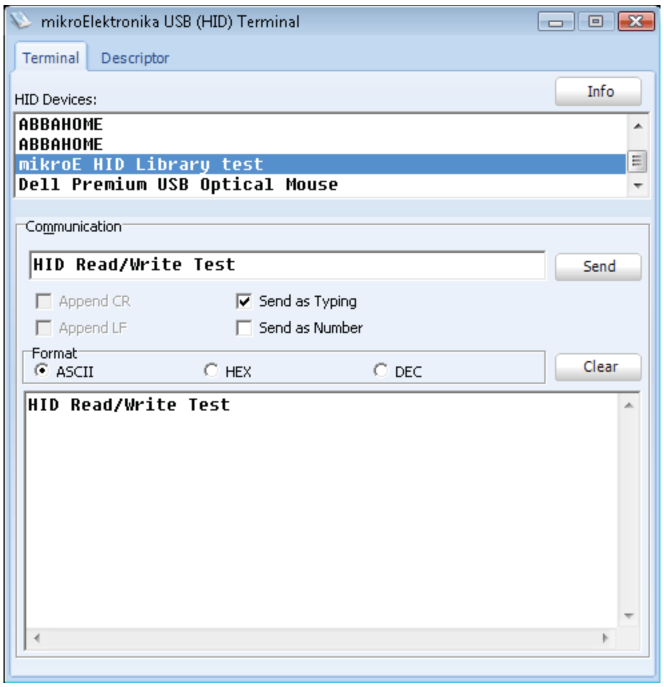
Graphic Lcd Bitmap Editor

The mikroPascal PRO for PIC32 includes the Graphic Lcd Bitmap Editor. Output is the mikroPascal PRO for PIC32 compatible code. You can launch it from the drop-down menu **Tools › Glcd Bitmap Editor**.



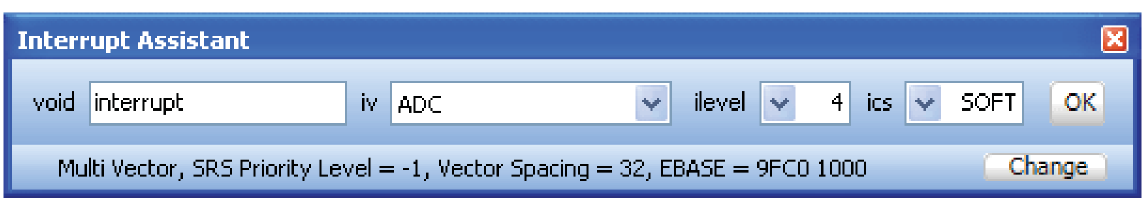
HID Terminal

The mikroPascal PRO for PIC32 includes the HID communication terminal for USB communication. You can launch it from the drop-down menu **Tools › HID Terminal**.



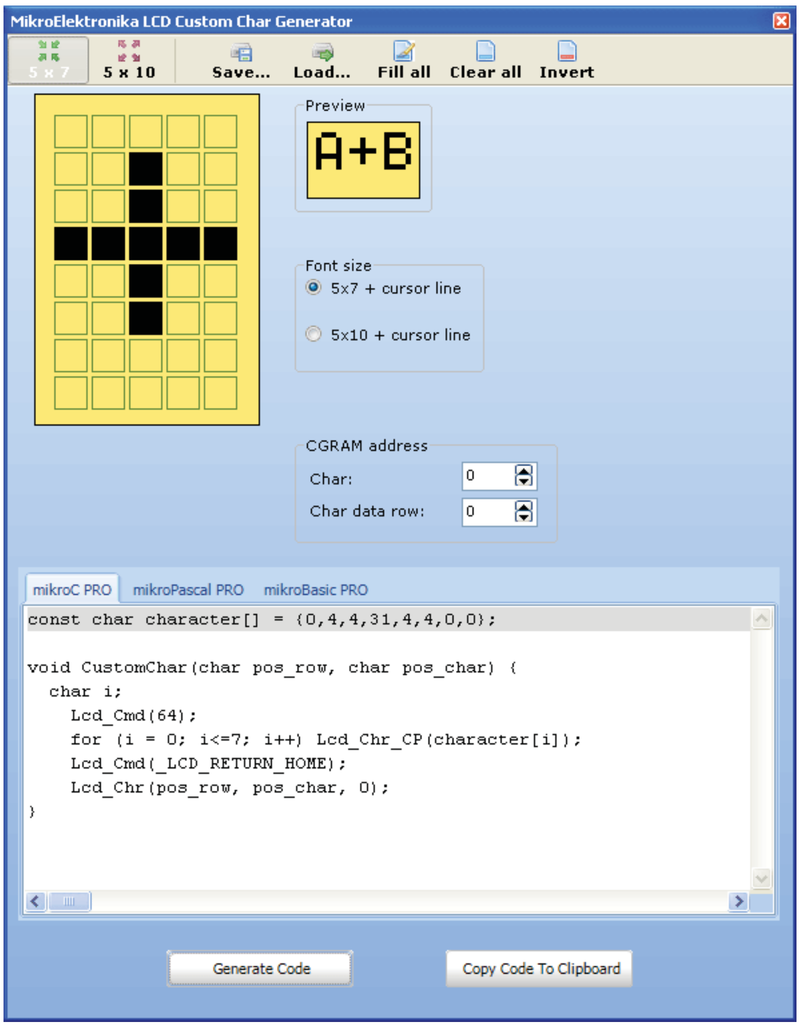
Interrupt Assistant

mikroPascal PRO for PIC32 includes the Interrupt Assistant that assist user in configuring interrupts. Output is the code for the configured interrupt routine. You can launch it from the drop-down menu **Tools › Interrupt Assistant**.



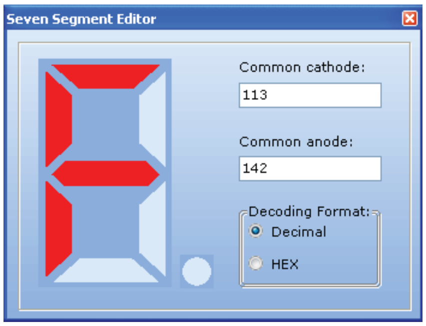
Lcd Custom Character

mikroPascal PRO for PIC32 includes the Lcd Custom Character. Output is mikroPascal PRO for PIC32 compatible code. You can launch it from the drop-down menu **Tools › Lcd Custom Character**.



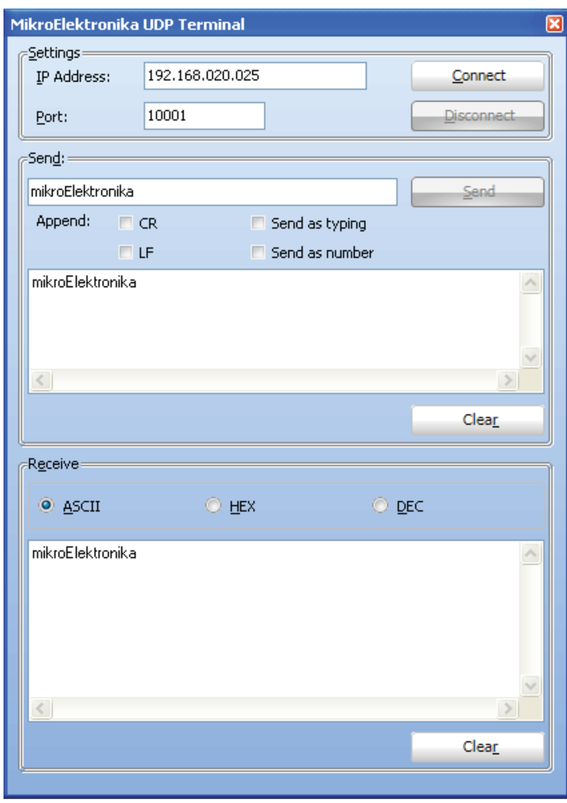
Seven Segment Editor

The Seven Segment Editor is a convenient visual panel which returns decimal/hex value for any viable combination you would like to display on seven segment display. Click on the parts of seven segment image to get the requested value in the edit boxes. You can launch it from the drop-down menu **Tools › Seven Segment Editor** or by clicking the Seven Segment Editor Icon  from Tools toolbar.



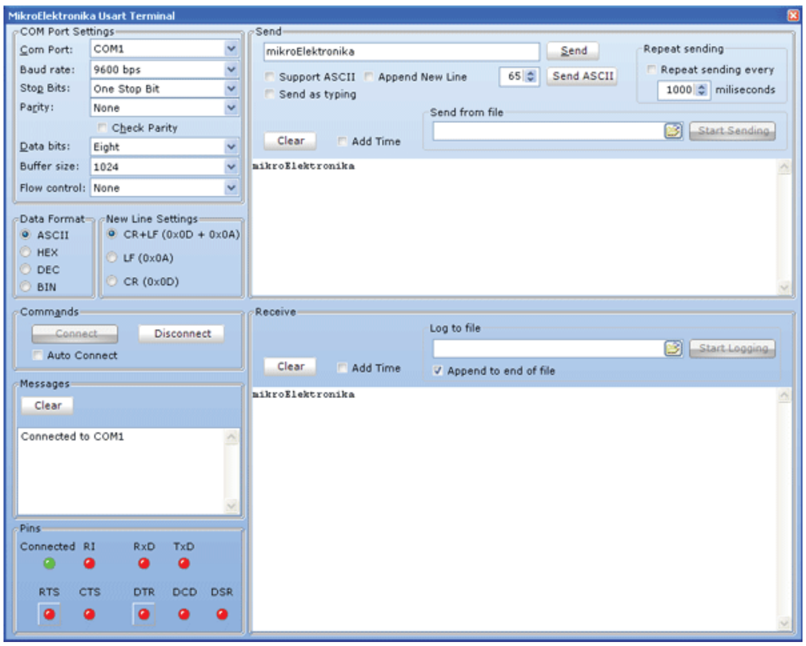
UDP Terminal

The mikroPascal PRO for PIC32 includes the UDP Terminal. You can launch it from the drop-down menu **Tools › UDP Terminal**.



USART Terminal

The mikroPascal PRO for PIC32 includes the USART communication terminal for RS232 communication. You can launch it from the drop-down menu **Tools › USART Terminal** or by clicking the USART Terminal Icon from Tools toolbar.



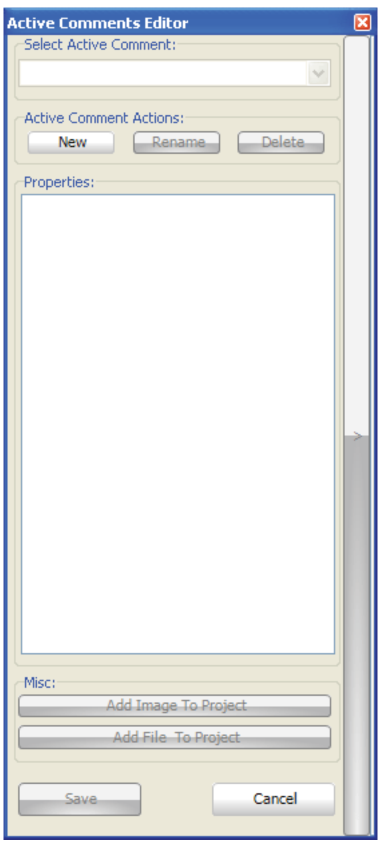
Active Comments

The idea of Active Comments is to make comments *alive* and give old fashioned comments new meaning and look. From now on, you can assign mouse event on your comments and 'tell' your comments what to do on each one. For example, on left mouse click, open some web address in your browser, on mouse over show some picture and on mouse double click open some file.

Suppose we are writing a example for a GSM/GPSR module which is connected to the EasyPIC6 and we would like to provide a photo of our hardware (jumpers, cables, etc.). within the example.
It would also be nice to put some documentation about chip we are using and a GSM module extra board. Now we can have all those things defined in one single comment using **Active Comment Editor**.

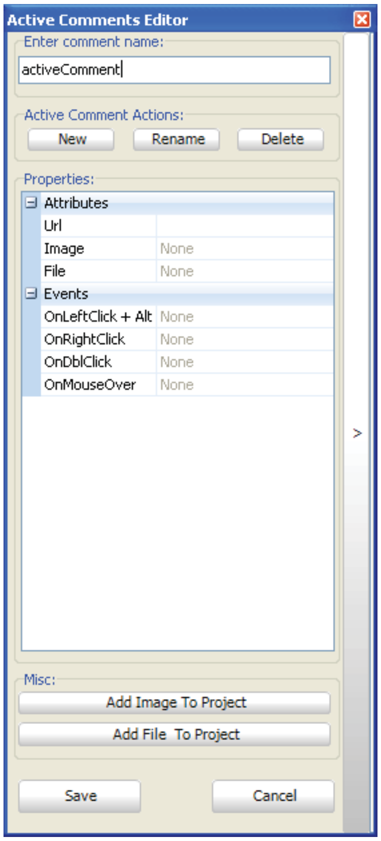
New Active Comment

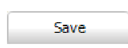
When you start Active Comment Editor for the first time (from the View menu, from editor's pop-up menu, or by pressing Ctrl + Alt + P) you will get an empty editor:



By clicking the 

button you are prompted to enter a name for the comment:



You can notice that when you start typing a name, properties pane is automatically displayed so you can edit properties if you wish. A Comment will be is created when you click 

button.

Properties are consisted of two major categories - Attributes and Events.

Attributes can be:

- URL - Valid web address.
- Image - Image has to be previously added to Project (Project Manager > Images).
- File - File has to be previously added to Project (Project Manager > Other Files).

There are four predefined event types you can apply to an Active Comment:

1. OnLeftClick + Alt
2. OnRightClick
3. OnDoubleClick
4. OnMouseOver

First three event types can have one of the following three actions:

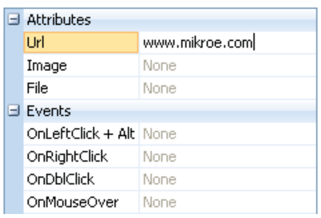
1. OpenUrl - Opens entered URL in default Web browser.
2. OpenFile - Opens a file within a default program associated with the file extension (defined by Windows).
3. None - Does nothing.

The fourth event, OnMouseOver, has only 2 actions:

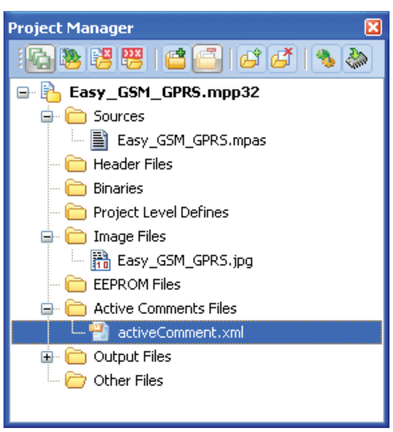
1. PreviewImage - Shows image when cursor is moved over a comment.
2. None - Does nothing.

Attributes are tightly bounded with events. For example, you can not have OnLeftClick + Alt -> OpenFile if there is no file attribute set, or if there is no file added to project. The same behavior applies to image attribute.

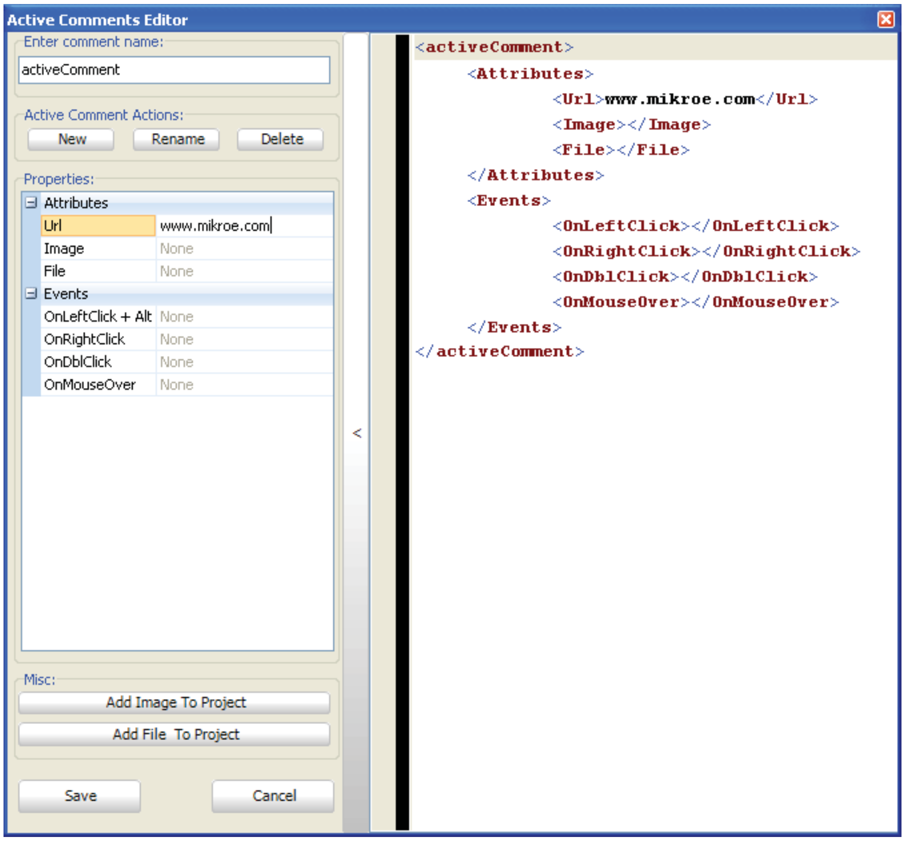
Let's start editing our Active Comment by entering some valid web address in the URL field:



For every Active Comment a XML file will be created, containing all valid information regarding the Active Comment - attributes, events, etc. and it is automatically added to Project manager after saving it:

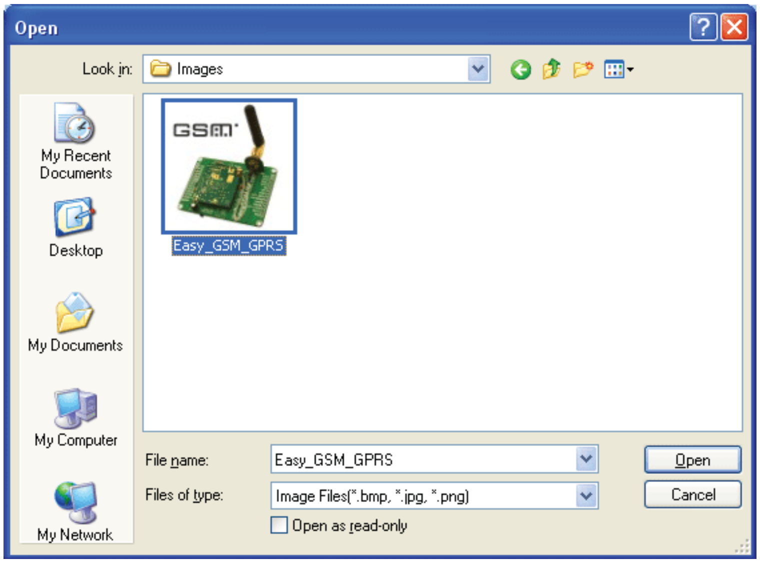


You can see the contents of the created XML file by expanding Active Comment Editor:



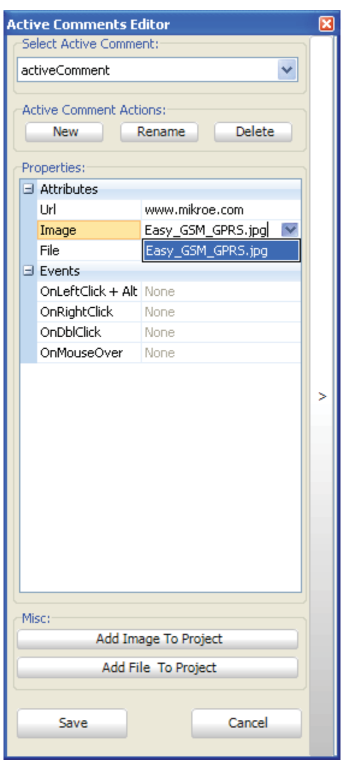
As we mentioned above you can add image or file which are already included in project. If the the desired image or file aren't added, you can do it directly from here by clicking the Add Image To Project or Add File To Project button.

Next file dialog will be opened:



There, you should select the desired image to be added. In our example, `Easy_GSM_GPRS.jpg` image will be added.

Selected picture is automatically added to the drop down list of the Image field in Active Comment Editor:



Now, when image has been selected, we can assign an event to it. For example, OnMouseOver will be used for PreviewImage action, and OnLeftClick + Alt will be assigned to OpenUrl action:

⊟ Attributes	
Url	www.mikroe.com
Image	Easy_GSM_GPRS.jpg
File	None
⊟ Events	
OnLeftClick + Alt	OpenUrl
OnRightClick	None
OnDblClick	None
OnMouseOver	PreviewImage

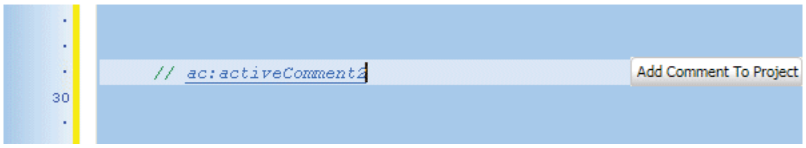
Now we can save our changes to Active Comment by clicking the Save button.

Note: Setting file attributes is same as for image, so it won't be explained separately.

Once we have finished creating our active comment, we can notice that it has been added to source file on current caret position with `ac`: prefix 'telling' IDE that it is active comment:

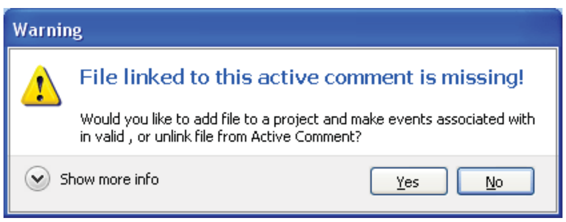
Now let's try it. If you LeftClick+Alt on it, URL in default Web browser will be opened. If you hover the mouse over it, you will see an Image preview:

There is another way to add an active comment to an active project. You can do it simply by typing a comment in old fashion way, except with `ac`: prefix. So it would look like this:

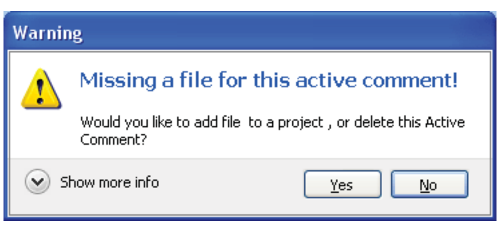


Notice that when you stop typing, Add Comment To Project button will show. By clicking on it, you will open Active Comment Editor and comment name will be already set, so you need only to adjust attributes and settings.
After saving you can always edit your active comment by Active Comment Editor, and switch between comments directly from editor.

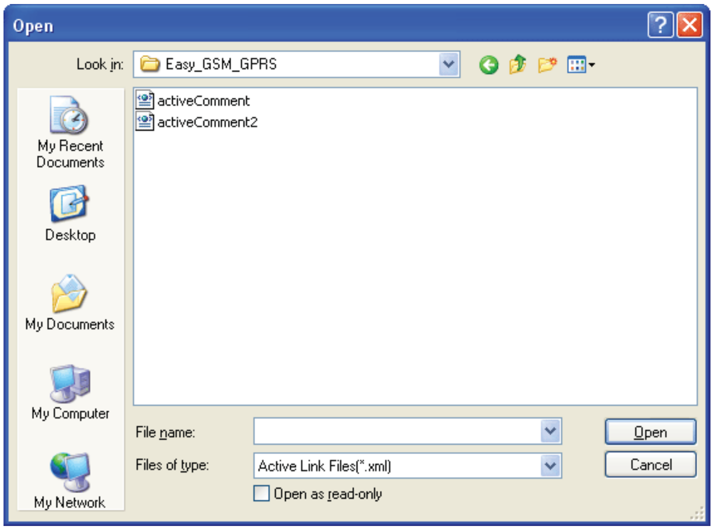
If you remove a file from the Project Manager or add an Active Comment File which contains information about the file which is no longer in project, and hover the mouse over the comment, you will be prompted to either add file to project or remove event definition from Active Comment for this file:



If you remove active comment file from the Project Manager, you'll receive this message:



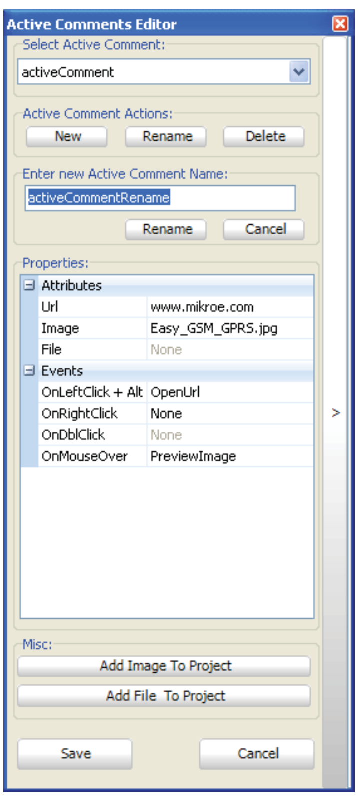
Click on Yes button you'll prompted for an active comment file:



If you click No, comment will be removed from the source code.

Renaming Active Comment

When you click on rename button, you will be prompted to enter new name:



Now click again Rename button. Now you have renamed your Active Comment in such a way that its filename, source code name are changed:

Deleting Active Comment

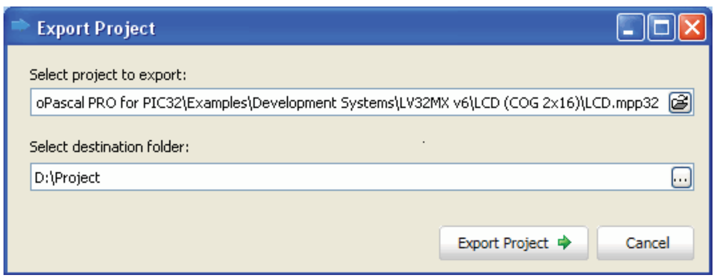
Deleting active comment works similar like renaming it. By clicking on delete button, you will remove an active comment from both code and Project Manager.

Export Project

This option is very convenient and finds its use in relocating your projects from one place to another (e.g. from your work computer to your home computer).

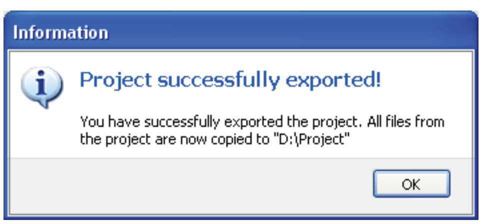
Often, project contains complicated search paths (files involved within your project could be in a different folders, even on different hard disks), so it is very likely that some files will be forgotten during manual relocation.
In order to simplify this, Export Project gives you opportunity to do this task automatically.

To open Export Project, from Project menu select Export Project or hit Ctrl + Alt + E. The following window will appear:



In the empty input boxes, current location and the destination folder of the desired project should be entered. By default, currently active project will be set for export. You can change it any time by clicking the Open Button  .

Once you have entered the appropriate data, click Export Project button. After exporting is done, and if everything was OK, you'll receive a message:



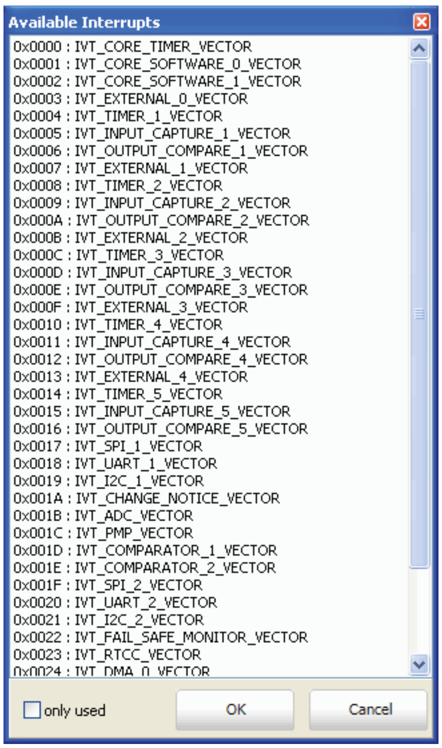
Now, Export Project has copied all project files into desired folder and changed project search paths, so you can easily move the entire folder to another location and run the project.

Jump To Interrupt

Lets you choose which interrupt you want to jump to.

Requirement: Interrupt routine is included in project.

You can call Jump To Interrupt by selecting **Run › Jump To Interrupt** from the drop-down menu, or by clicking the Jump To Interrupt Icon  , from the Watch Values Window.



By checking the Only Used box, you can display only the used breakpoints.

Regular Expressions

Introduction

Regular Expressions are a widely-used method of specifying patterns of text to search for. Special metacharacters allow you to specify, for instance, that a particular string you are looking for, occurs at the beginning, or end of a line, or contains `n` recurrences of a certain character.

Simple matches

Any single character matches itself, unless it is a metacharacter with a special meaning described below. A series of characters matches that series of characters in the target string, so the pattern "`short`" would match "`short`" in the target string. You can cause characters that normally function as metacharacters or escape sequences to be interpreted by preceding them with a backslash "`\`".
For instance, metacharacter "`^`" matches beginning of string, but "`\^`" matches character "`^`", and "`\\`" matches "`\`", etc.

Examples:

`unsigned` matches string **`'unsigned'`**
`\^unsigned` matches string **`'^unsigned'`**

Escape sequences

Characters may be specified using an escape sequences: "**`\n`**" matches a newline, "**`\t`**" a tab, etc. More generally, **`\xnn`**, where **`nn`** is a string of hexadecimal digits, matches the character whose ASCII value is **`nn`**.
If you need wide (Unicode) character code, you can use '**`\x{nnnn}`**', where '**`nnnn`**' - one or more hexadecimal digits.

`\xnn` - char with hex code **`nn`**
`\x{nnnn)`- char with hex code **`nnnn`** (one byte for plain text and two bytes for Unicode)
`\t` - tab (HT/TAB), same as **`\x09`**
`\n` - newline (NL), same as **`\x0a`**
`\r` - car.return (CR), same as **`\x0d`**
`\f` - form feed (FF), same as **`\x0c`**
`\a` - alarm (bell) (BEL), same as **`\x07`**
`\e` - escape (ESC) , same as **`\x1b`**

Examples:

`unsigned\x20int` matches **`'unsigned int'`** (note space in the middle)
`\tunsigned` matches **`'unsigned'`** (predecessed by tab)

Character classes

You can specify a character class, by enclosing a list of characters in **`[]`**, which will match any of the characters from the list. If the first character after the "**`[`**" is "**`^`**"**,** the class matches any character not in the list.

Examples:

`count[aeiou]r` finds strings `'countar'`, `'counter'`, etc. but not `'countbr'`, `'countcr'`, etc.
`count[^aeiou]r` finds strings `'countbr'`, `'countcr'`, etc. but not `'countar'`, `'counter'`, etc.

Within a list, the `"-"` character is used to specify a range, so that `a-z` represents all characters between `"a"` and `"z"`, inclusive.

If you want `"-"` itself to be a member of a class, put it at the start or end of the list, or precede it with a backslash.
If you want `']'`, you may place it at the start of list or precede it with a backslash.

Examples:

`[-az]` matches `'a'`, `'z'` and `'-'`
`[az-]` matches `'a'`, `'z'` and `'-'`
`[a\-z]` matches `'a'`, `'z'` and `'-'`
`[a-z]` matches all twenty six small characters from `'a'` to `'z'`
`[\n-\x0D]` matches any of `#10,#11,#12,#13`.
`[\d-t]` matches any digit, `'-'` or `'t'`.
`[]-a]` matches any char from `']'`..`'a'`.

Metacharacters

Metacharacters are special characters which are the essence of regular expressions. There are different types of metacharacters, described below.

Metacharacters - Line separators

`^` - start of line
`$` - end of line
`\A` - start of text
`\Z` - end of text
`.` - any character in line

Examples:

`^PORTA` - matches string ' `PORTA` ' only if it's at the beginning of line
`PORTA$` - matches string ' `PORTA` ' only if it's at the end of line
`^PORTA$` - matches string ' `PORTA` ' only if it's the only string in line
`PORT.r` - matches strings like '`PORTA`', '`PORTB`', '`PORT1`' and so on

The `"^"` metacharacter by default is only guaranteed to match beginning of the input string/text, and the `"$"` metacharacter only at the end. Embedded line separators will not be matched by `^"` or `"$"`.
You may, however, wish to treat a string as a multi-line buffer, such that the `"^"` will match after any line separator within the string, and `"$"` will match before any line separator.
Regular expressions works with line separators as recommended at http://www.unicode.org/unicode/reports/tr18/

Metacharacters - Predefined classes

\w - an alphanumeric character (including "_")
\W - a nonalphanumeric character
\d - a numeric character
\D - a non-numeric character
\s - any space (same as [\t\n\r\f])
\S - a non space

You may use \w, \d and \s within custom character classes.

Example:

routi\de - matches strings like 'routi1e', 'routi6e' and so on, but not 'routine', 'routime' and so on.

Metacharacters - Word boundaries

A word boundary ("\b") is a spot between two characters that has an alphanumeric character ("\w") on one side, and a nonalphanumeric character ("\W") on the other side (in either order), counting the imaginary characters off the beginning and end of the string as matching a "\W".

\b - match a word boundary)
\B - match a non-(word boundary)

Metacharacters - Iterators

Any item of a regular expression may be followed by another type of metacharacters - iterators. Using this metacharacters,you can specify number of occurences of previous character, metacharacter or subexpression.

* - zero or more ("greedy"), similar to {0,}
+ - one or more ("greedy"), similar to {1,}
? - zero or one ("greedy"), similar to {0,1}
{n} - exactly n times ("greedy")
{n,} - at least n times ("greedy")
{n,m} - at least n but not more than m times ("greedy")
*? - zero or more ("non-greedy"), similar to {0,}?
+? - one or more ("non-greedy"), similar to {1,}?
?? - zero or one ("non-greedy"), similar to {0,1}?
{n}? - exactly n times ("non-greedy")
{n,}? - at least n times ("non-greedy")
{n,m}? - at least n but not more than m times ("non-greedy")

So, digits in curly brackets of the form, {n,m}, specify the minimum number of times to match the item n and the maximum m. The form {n} is equivalent to {n,n} and matches exactly n times. The form {n,} matches n or more times. There is no limit to the size of n or m, but large numbers will chew up more memory and slow down execution.

If a curly bracket occurs in any other context, it is treated as a regular character.

Examples:

`count.*r ß`- matches strings like `'counter'`, `'countelkjdflkj9r'` and `'countr'`
`count.+r` - matches strings like `'counter'`, `'countelkjdflkj9r'` but not `'countr'`
`count.?r` - matches strings like `'counter'`, `'countar'` and `'countr'` but not `'countelkj9r'`
`counte{2}r` - matches string `'counteer'`
`counte{2,}r` - matches strings like `'counteer'`, `'counteeer'`, `'counteeer'` etc.
`counte{2,3}r` - matches strings like `'counteer'`, or `'counteeer'` but not `'counteeeer'`

A little explanation about "greediness". "Greedy" takes as many as possible, "non-greedy" takes as few as possible. For example, `'b+'` and `'b*'` applied to string `'abbbbc'` return `'bbbb'`, `'b+?'` returns `'b'`, `'b*?'` returns empty string, `'b{2,3}?'` returns `'bb'`, `'b{2,3}'` returns `'bbb'`.

Metacharacters - Alternatives

You can specify a series of alternatives for a pattern using `"|"` to separate them, so that `bit|bat|bot` will match any of `"bit"`, `"bat"`, or `"bot"` in the target string as would `"b(i|a|o)t)"`. The first alternative includes everything from the last pattern delimiter `("(", "[",` or the beginning of the pattern) up to the first `"|"`, and the last alternative contains everything from the last `"|"` to the next pattern delimiter. For this reason, it's common practice to include alternatives in parentheses, to minimize confusion about where they start and end.

Alternatives are tried from left to right, so the first alternative found for which the entire expression matches, is the one that is chosen. This means that alternatives are not necessarily greedy. For example: when matching `rou|rout` against `"routine"`, only the `"rou"` part will match, as that is the first alternative tried, and it successfully matches the target string (this might not seem important, but it is important when you are capturing matched text using parentheses.) Also remember that "|" is interpreted as a literal within square brackets, so if you write `[bit|bat|bot]`, you're really only matching `[biao|]`.

Examples:

`rou(tine|te)` - matches strings `'routine'` or `'route'`.

Metacharacters - Subexpressions

The bracketing construct `( ... )` may also be used for define regular subexpressions. Subexpressions are numbered based on the left to right order of their opening parenthesis. The first subexpression has number `'1'`

Examples:

`(int){8,10}` matches strings which contain 8, 9 or 10 instances of the `'int'`
`routi([0-9]|a+)e` matches `'routi0e'`, `'routi1e'` , `'routine'`, `'routinne'`, `'routinnne'` etc.

Metacharacters - Backreferences

Metacharacters `\1` through `\9` are interpreted as backreferences. `\` matches previously matched subexpression `#`.

Examples:

`(.)\1+` matches `'aaaa'` and `'cc'`.
`(.+)\1+` matches `'abab'` and `'123123'`
`(['"]?)(\d+)\1` matches `"13"` (in double quotes), or `'4'` (in single quotes) or `77` (without quotes) etc.

Keyboard Shortcuts

Below is a complete list of keyboard shortcuts available in mikroPascal PRO for PIC32 IDE.

IDE Shortcuts	
F1	Help
Ctrl+N	New Unit
Ctrl+O	Open
Ctrl+Shift+O	Open Project
Ctrl+Shift+N	New Project
Ctrl+K	Close Project
Ctrl+F4	Close unit
Ctrl+Shift+E	Edit Project
Ctrl+F9	Build
Shift+F9	Build All
Ctrl+F11	Build And Program
Shift+F4	View Breakpoints
Ctrl+Shift+F5	Clear Breakpoints
F11	Start mE Programmer
Ctrl+Shift+F11	Project Manager
F12	Options
Alt + X	Close mikroPascal PRO for PIC32
Basic Editor Shortcuts	
F3	Find, Find Next
Shift+F3	Find Previous
Alt+F3	Grep Search, Find In Files
Ctrl+A	Select All
Ctrl+C	Copy
Ctrl+F	Find
Ctrl+R	Replace
Ctrl+P	Print
Ctrl+S	Save Unit
Ctrl+Shift+S	Save All
Ctrl+V	Paste
Ctrl+X	Cut
Ctrl+Y	Delete Entire Line
Ctrl+Z	Undo
Ctrl+Shift+Z	Redo

Advanced Editor Shortcuts	
Ctrl+Space	Code Assistant
Ctrl+Shift+Space	Parameters Assistant
Ctrl+D	Find Declaration
Ctrl+E	Incremental Search
Ctrl+L	Routine List
Ctrl+G	Goto Line
Ctrl+J	Insert Code Template
Ctrl+Shift+.	Comment Code
Ctrl+Shift+,	Uncomment Code
Ctrl+*number*	Goto Bookmark
Ctrl+Shift+*number*	Set Bookmark
Ctrl+Shift+I	Indent Selection
Ctrl+Shift+U	Unindent Selection
TAB	Indent Selection
Shift+TAB	Unindent Selection
Alt+Select	Select Columns
Ctrl+Alt+Select	Select Columns
Alt + Left Arrow	Fold Region (if available)
Alt + Right Arrow	Unfold Region (if available)
Ctrl+Alt+L	Convert Selection to Lowercase
Ctrl+Alt+U	Convert Selection to Uppercase
Ctrl+Alt+T	Convert to Titlecase
Ctrl+T	USART Terminal
Ctrl+Q	Quick Converter
mikroICD Debugger and Software Simulator Shortcuts	
F2	Jump To Interrupt
F4	Run to Cursor
F5	Toggle Breakpoint
F6	Run/Pause Debugger
F7	Step Into
F8	Step Over
F9	Start Debugger
Ctrl+F2	Stop Debugger

Ctrl+F5	Add to Watch List
Ctrl+F8	Step Out
Alt+D	Disassembly View
Shift+F5	Open Watch Window
Ctrl+Shift+A	Show Advanced Breakpoints

CHAPTER 3

mikroPascal PRO for PIC32 Command Line Options

Usage: `mPPIC32.exe [-<opts> [-<opts>]] [<infile> [-<opts>]] [-<opts>]]`
Infile can be of *.c, *.emcl and *.pld type.

The following parameters are valid:

- `-P <devicename>` : MCU for which compilation will be done.
- `-FO <oscillator>` : Set oscillator [in MHz].
- `-SP <directory>` : Add directory to the search path list.
- `-N <filename>` : Output files generated to file path specified by filename.
- `-B <directory>` : Save compiled binary files (`*.emcl`) to 'directory'.
- `-O` : Miscellaneous output options.
- `-DBG` : Generate debug info.
- `-MSF` : Short message format.
- `-Y` : Dynamic link for string literals.
- `-RA` : Rebuild all sources in project.
- `-L` : Check and rebuild new libraries.
- `-DL` : Build all files as libraries.
- `-LHF` : Generate Long hex format.
- `-PF` : Project file name.
- `-EH <filename>` : Full EEPROM HEX file name with path.
- `-HEAP <size>` : Heap size in bytes.
- `-GC` : Generate COFF file.
- `-PF` : Project file name.
- `-SSA` : Enable SSA optimization.
- `-UICD` : ICD build type.
- `-INTDEF` : Interrupt settings.
- `-EBASE` : Exception base address.

Example:

```
mPPIC32.exe -MSF -DBG -p32MX460F512L -Y -DL -O11111114 -fo80 -N"C:\Lcd\Lcd.mpp32" 
-SP"C:\Program Files\Mikroelektronika\mikroPascal PRO for PIC32\Defs" 
                -SP"C:\Program Files\Mikroelektronika\mikroPascal PRO for PIC32\Uses" 
-SP"C:\Lcd\" "__Lib_Math.emcl" "__Lib_MathDouble.emcl" 
            "__Lib_System.emcl" "__Lib_Delays.emcl" "__Lib_LcdConsts.emcl" "__Lib_Lcd.
emcl" "Lcd.mpas"
```

Parameters used in the example:

`-MSF` : Short Message Format; used for internal purposes by IDE.
`-DBG` : Generate debug info.
`-p32MX460F512L` : MCU PIC32MX460F512L selected.
`-Y` : Dynamic link for string literals enabled.
`-DL` : All files built as libraries.
`-O11111114` : Miscellaneous output options.
`-fo80` : Set oscillator frequency [in MHz].
`-N"C:\Lcd\Lcd.mpp32" -SP"C:\Program Files\Mikroelektronika\mikroPascal PRO for PIC32\Defs"` : Output files generated to file path specified by filename.
`-SP"C:\Program Files\Mikroelektronika\mikroPascal PRO for PIC32\Defs"` : Add directory to the search path list.
`-SP"C:\Program Files\Mikroelektronika\mikroPascal PRO for PIC32\Uses"` : Add directory to the search path list.
`-SP"C:\Lcd\"` : Add directory to the search path list.
`"Lcd.mpas" "__Lib_Math.emcl" "__Lib_MathDouble.emcl" "__Lib_System.emcl" "__Lib_Delays.emcl" "__Lib_LcdConsts.emcl" "__Lib_Lcd.emcl"` : Specify input files.

CHAPTER 4

mikroICD (In-Circuit Debugger)

Introduction

The mikroICD is a highly effective tool for a **Real-Time debugging** on hardware level. The mikroICD debugger enables you to execute the mikroPascal PRO for PIC32 program on a host PIC32 microcontroller and view variable values, Special Function Registers (SFR), RAM, CODE and EEPROM memory along with the mikroICD code execution on hardware.

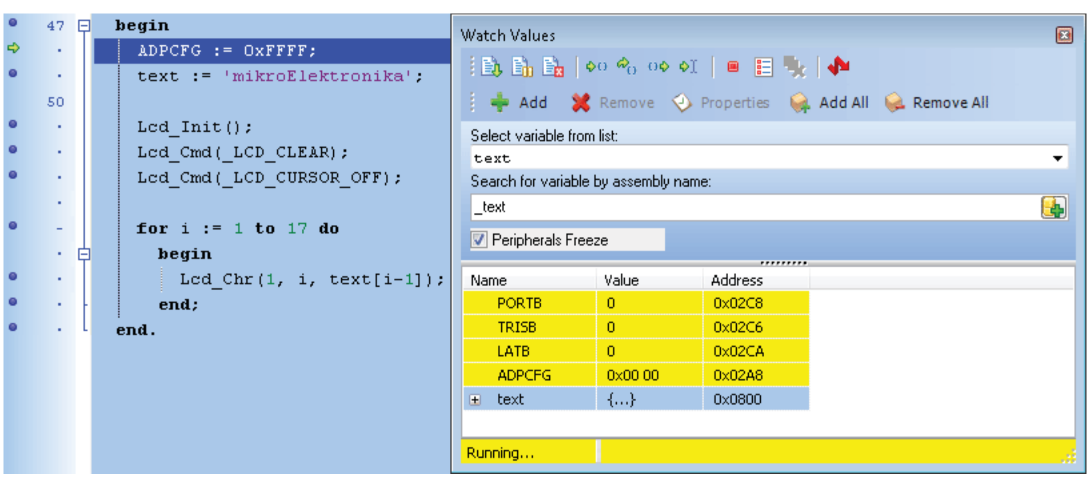
Step No. 1

If you have appropriate hardware and software for using the mikroICD select **mikroICD Debug** Build Type before compiling the project.

Now, compile the project by pressing Ctrl + F9, or by pressing Build Icon  on Build Toolbar.

Step No. 2

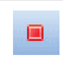
Run the mikroICD by selecting **Run › Start Debugger** from the drop-down menu or by clicking the Start Debugger Icon  . Starting the Debugger makes more options available: Step Into, Step Over, Run to Cursor, etc. Line that is to be executed is color highlighted (blue by default). There is also notification about the program execution and it can be found in the Watch Window (yellow status bar). Note that some functions take more time to execute; execution is indicated with "Running..." message in the Watch Window Status Bar.



Related topics: mikroICD Debugger Example, mikroICD Debug Windows, mikroICD Debugger Options

mikroICD Debugger Options

Debugger Options

Name	Description	Function Key	Toolbar Icon
Start Debugger	Starts Debugger.	F9	
Stop Debugger	Stop Debugger.	Ctrl + F2	
Run/Pause Debugger	Run/Pause Debugger.	F6	
Step Into	Executes the current program line, then halts. If the executed program line calls another routine, the debugger steps into the routine and halts after executing the first instruction within it.	F7	
Step Over	Executes the current program line, then halts. If the executed program line calls another routine, the debugger will not step into it. The whole routine will be executed and the debugger halts at the first instruction following the call.	F8	
Step Out	Executes all remaining program lines within the subroutine. The debugger halts immediately upon exiting the subroutine.	F8	
Run To Cursor	Executes the program until reaching the cursor position.	Ctrl + F8	
Toggle Breakpoint	Toggle breakpoints option sets new breakpoints or removes those already set at the current cursor position.	F5	

Related topics: Run Menu, Debug Toolbar

mikroICD Debugger Example

Here is a step-by-step mikroICD Debugger Example.

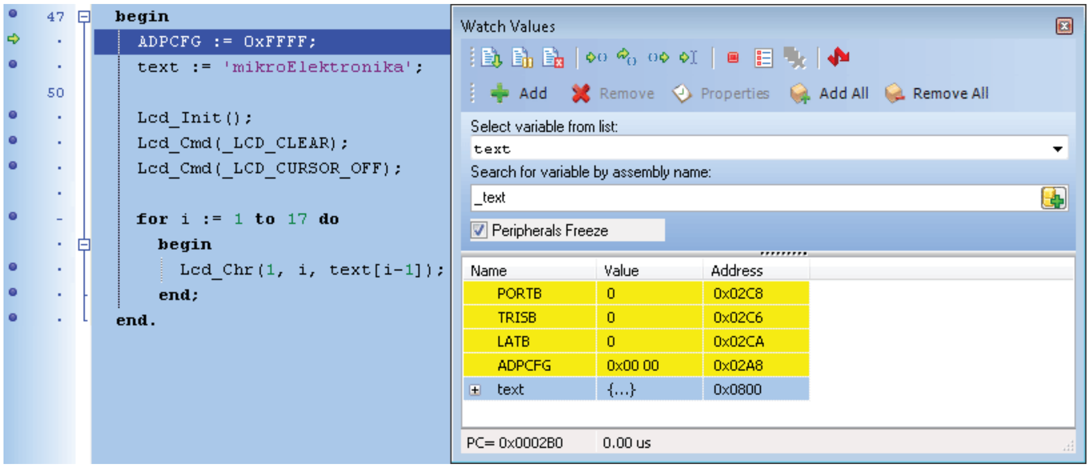
Step No. 1

First you have to write a program. We will show how the mikroICD works using this example:

```
program  Lcd_Test;

// LCD module connections
var LCD_RS : sbit at LATD0_bit;
var LCD_EN : sbit at LATD1_bit;
var LCD_D4 : sbit at LATB0_bit;
var LCD_D5 : sbit at LATB1_bit;
var LCD_D6 : sbit at LATB2_bit;
var LCD_D7 : sbit at LATB3_bit;

var LCD_RS_Direction : sbit at TRISD0_bit;
var LCD_EN_Direction : sbit at TRISD1_bit;
var LCD_D4_Direction : sbit at TRISB0_bit;
var LCD_D5_Direction : sbit at TRISB1_bit;
var LCD_D6_Direction : sbit at TRISB2_bit;
var LCD_D7_Direction : sbit at TRISB3_bit;
// End LCD module connections

var text : array[16] of char;
    i    : byte;

  begin
    ADPCFG := 0xFFFF;
    text := 'mikroElektronika';

    Lcd_Init();
    Lcd_Cmd(_LCD_CLEAR);
    Lcd_Cmd(_LCD_CURSOR_OFF);

    for i := 1 to 17 do
        Lcd_Chr(1, i, text[i-1]);
  end.
```

Step No. 2

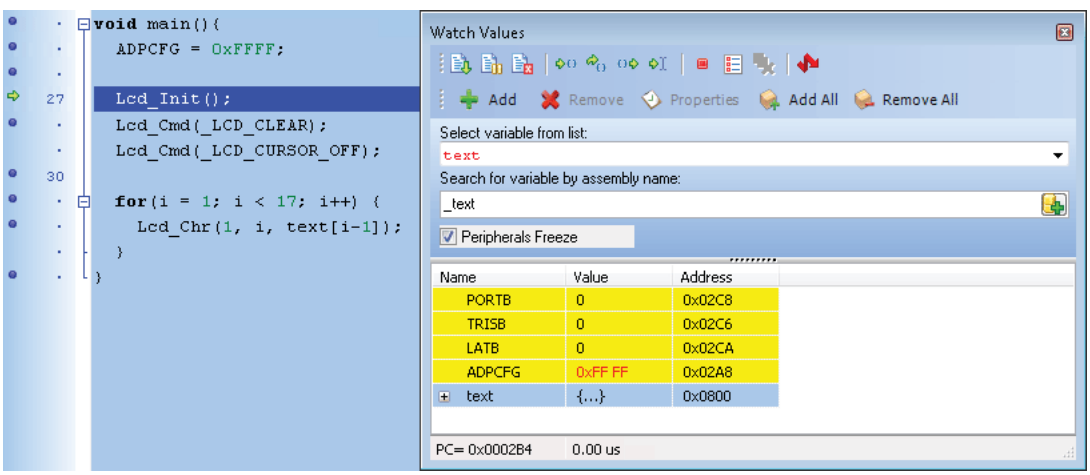
After successful compilation and MCU programming press **F9** to start the mikroICD. After the mikroICD initialization a blue active line should appear.



Step No. 3

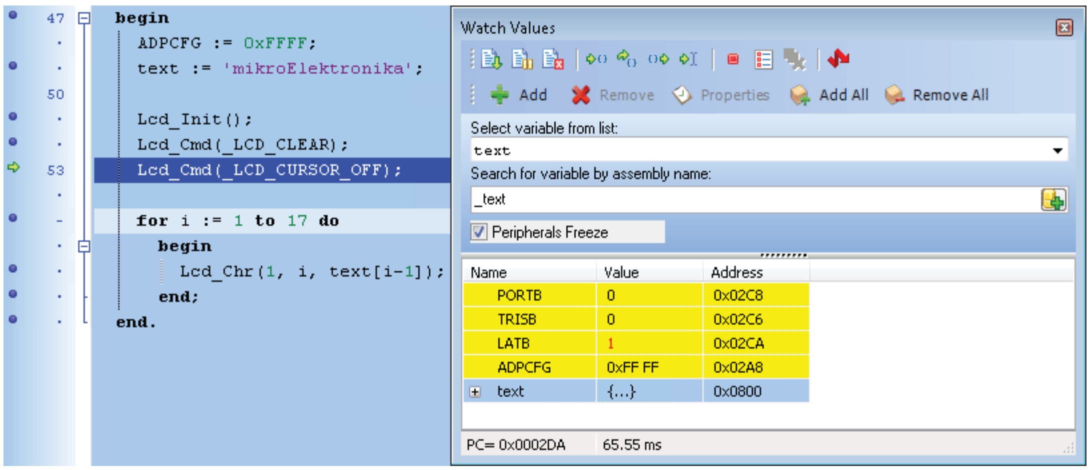
We will debug the program line by line. Pressing **[F8]** we are executing code line by line. However, it is not recommended that user does not use Step Into **[F7]** and Step Over **[F8]** over Delays routines and routines containing delays. Instead use Run to cursor **[F4]** and Breakpoints functions.

All changes are read from MCU and loaded into Watch Window. Note that **TRISB** changed its value from 255 to 0.



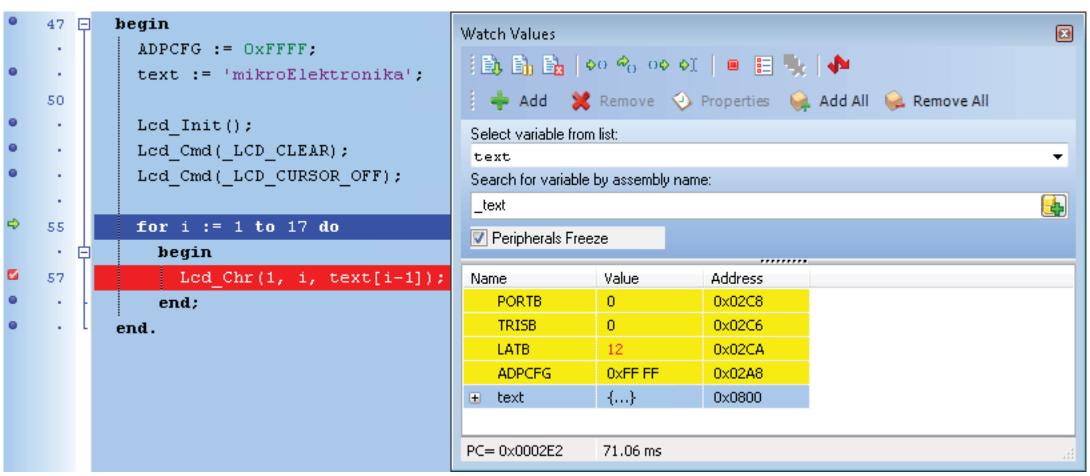
Step No. 4

Step Into [**F7**], Step Over [**F8**] and Step Out [**Ctrl+F8**] are mikroICD debugger functions that are used in stepping mode. There is also a Real-Time mode supported by the mikroICD. Functions that are used in the Real-Time mode are Run/ Pause Debugger [**F6**] and Run to cursor [**F4**]. Pressing **F4** executes the code until the program reaches the cursor position line.



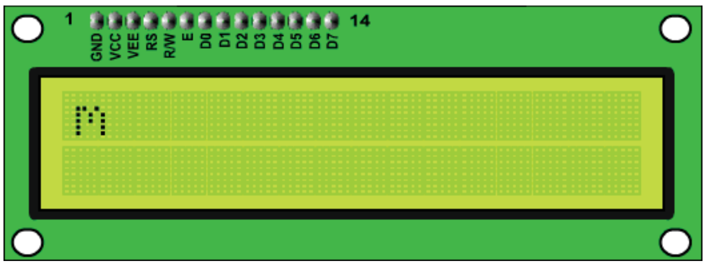
Step No. 5

Run(Pause) Debugger [**F6**] and Toggle Breakpoints [**F5**] are mikroICD debugger functions that are used in the Real-Time mode. Pressing **F5** marks the line selected by the user for breakpoint. **F6** executes code until the breakpoint is reached. After reaching the breakpoint Debugger halts. Here in our example we will use breakpoints for writing “mikroElektronika” on Lcd char by char. Breakpoint is set on Lcd_Chr and the program will stop everytime this function is reached. After reaching breakpoint we must press **F6** again to continue the program execution.



Breakpoints are divided into two groups: hardware and software breakpoints. The hardware breakpoints are placed in the MCU and they provide fastest debugging. Number of hardware breakpoints is limited to 8 (6 instruction, 2 data). If all hardware brekpoints are used, then the next breakpoint will be software breakpoint. These breakpoints are placed inside the mikroICD and simulate hardware breakpoints. Software breakpoints are much slower than hardware breakpoints. These differences between hardware and software breakpoints are not visible in the mikroICD software but their different timings are quite notable. That's why it is important to know that there are two types of breakpoints.

The picture below demonstrates step-by-step execution of the code used in above mentioned examples.



Common Errors:

- Trying to program the MCU while the mikroICD is active.
- Trying to debug **Release** build version of the program with the mikroICD debugger.
- Trying to debug program code which has been changed, but has not been compiled and programmed into the MCU.
- Trying to select line that is empty for Run to cursor [**F4**] and Toggle Breakpoints [**F5**] functions.
- Trying to debug MCU with mikroICD while Watch Dog Timer is enabled.
- Trying to debug MCU with mikroICD while Power Up Timer is enabled.
- Trying to **Step Into** [**F7**] the mikroPascal PRO for PIC32 Library routines. Use **Step Over** [**F8**] command for these routines.
- It is not possible to force Code Protect while trying to debug MCU with mikroICD.
- Trying to debug MCU with mikroICD with pull-up resistors set to ON on RB6 and RB7.

Related topics: mikroICD Debugger, mikroICD Debug Windows, mikroICD Debugger Options

mikroICD Debugger Windows

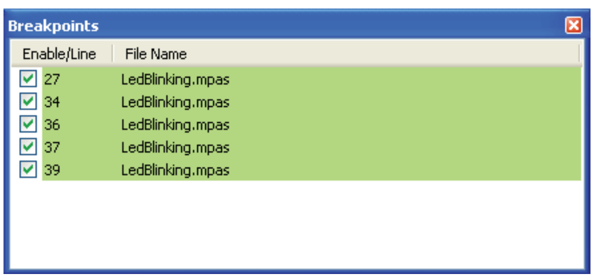
Debug Windows

This section provides an overview of available Debug Windows in mikroPascal PRO for PIC32:

- Breakpoints Window
- Watch Values Window
- RAM Window
- Stopwatch Window
- EEPROM Watch Window
- Code Watch Window

Breakpoints Window

The Breakpoints window manages the list of currently set breakpoints in the project. Doubleclicking the desired breakpoint will cause cursor to navigate to the corresponding location in source code.
In situations when multiple breakpoints are used within the code, it is sometimes handy to enable/disable certain breakpoints. To do this, just check/uncheck the desired breakpoint using the checkbox in front of the breakpoint's name.



Watch Values Window

Watch Values Window is the main Debugger window which allows you to monitor program execution. To show the Watch Values Window, select **Debug Windows › Watch** from the **View** drop-down menu.

The Watch Values Window displays variables and registers of the MCU, with their addresses and values. Values are updated along with the code execution. Recently changed items are coloured red.

There are two ways to add variable/register into the watch list:

- by its real name (variable's name in program code). Just select wanted variable/register from **Select variable from list** drop-down menu and click the 

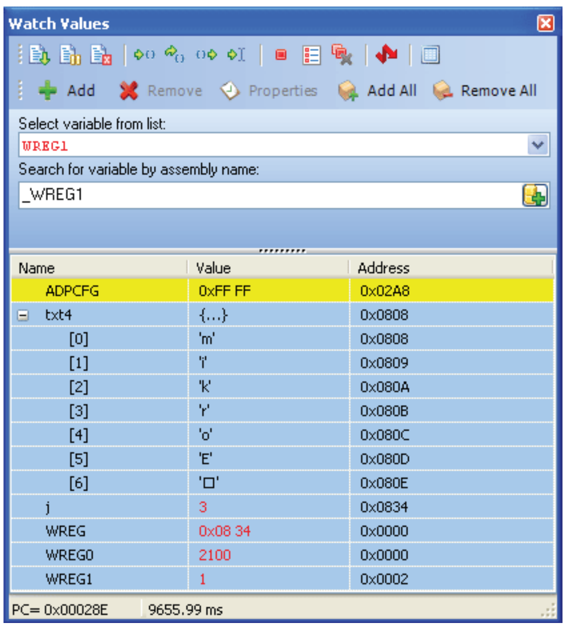
button.
- by its name ID (assembly variable name). Simply type name ID of the variable/register you want to display into **Search for variable by assemby name** box and click the 

button.

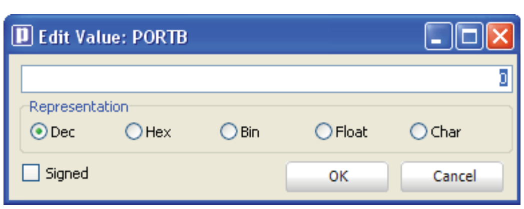
Also, it is possible to add all variables in the Watch Values Window by clicking Add All button.

To remove a variable from the Watch Values Window, just select the variable that you want to remove and then click the Remove button, or press the Delete key.
It is possible to remove all variables from the Watch Values Window by clicking Remove All button.

You can also expand/collapse complex variables i.e. struct type variables, strings, etc, by clicking the appropriate button (+ or −) beside variable name.



Double clicking a variable or clicking the Properties button opens the Edit Value window in which you can assign a new value to the selected variable/register. Also, you can choose the format of variable/register representation between decimal, hexadecimal, binary, float or character. All representations except float are unsigned by default. For signed representation click the check box next to the **Signed** label.



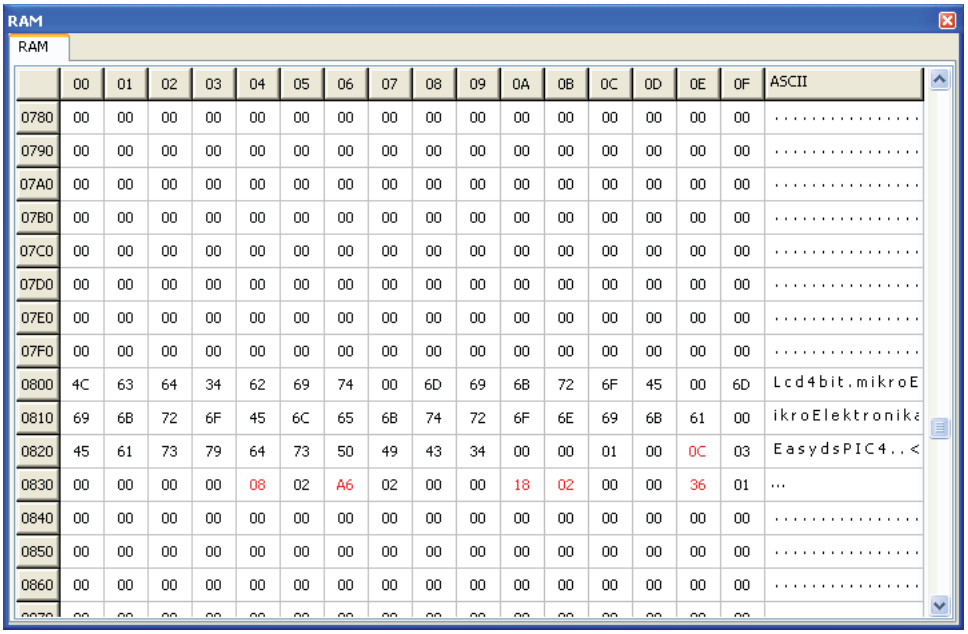
An item's value can also be changed by double clicking item's value field and typing the new value directly.

RAM Window

The RAM Window is available from the drop-down menu, **View › Debug Windows › RAM**.

The RAM Window displays the map of MCU's RAM, with recently changed items colored red. The user can edit and change the values in the RAM window.

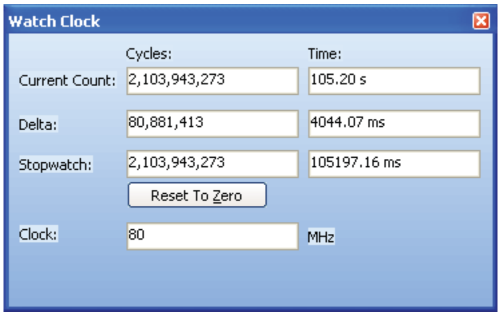
mikroICD Specific: RAM window content will be written to the MCU before the next instruction execution.



Stopwatch Window

The Software Simulator Stopwatch Window is available from the drop-down menu, **View › Debug Windows › Stopwatch**.

The Stopwatch Window displays a `Current Count` of cycles/time since the last Software Simulator action.
`Stopwatch` measures the execution time (number of cycles) from the moment Software Simulator has started and can be reset at any time.
`Delta` represents the number of cycles between the lines where Software Simulator action has started and ended.



Notes:

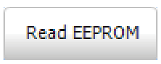
- The user can change the clock in the Stopwatch Window, which will recalculate values for the latest specified frequency.
- Changing the clock in the Stopwatch Window does not affect actual project settings – it only provides a simulation.
- Stopwatch is available only when Software Simulator is selected as a debugger.

EEPROM Watch Window

Note: EEPROM Watch Window is available only when mikroICD is selected as a debugger.

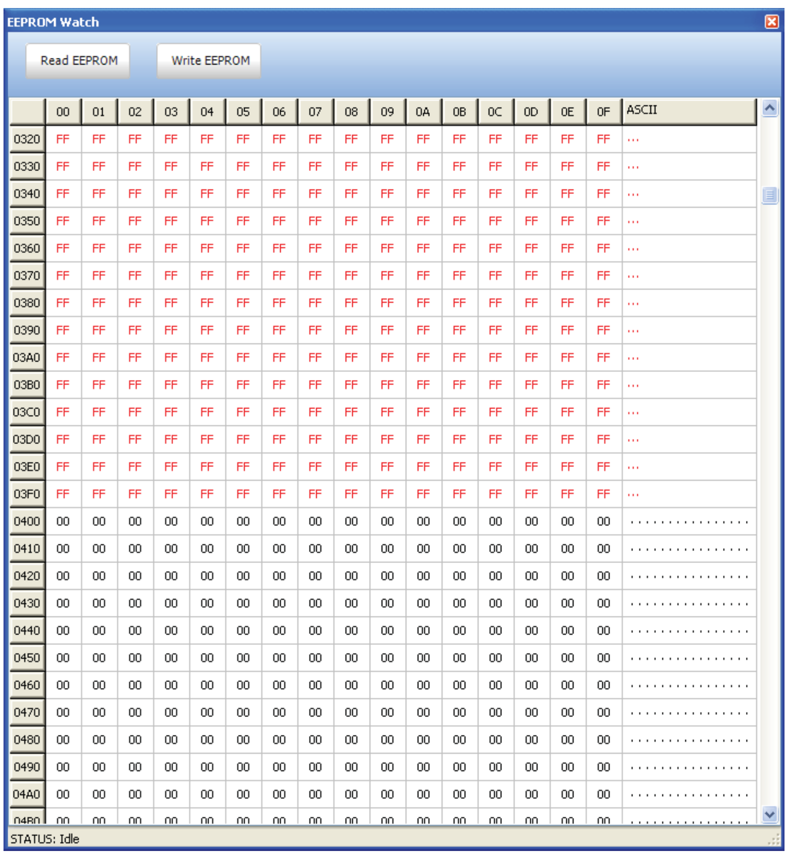
To show the EEPROM Watch Window, select **Debug Windows › EEPROM** from the **View** drop-down menu.
The EEPROM Watch Window shows current content of the MCU's internal EEPROM memory.

There are two action buttons concerning the EEPROM Watch Window:



- Reads data from MCU's internal EEPROM memory and loads it up into the EEPROM window.

- Writes data from the EEPROM window into MCU's internal EEPROM memory.



	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0320	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
0330	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
0340	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
0350	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
0360	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
0370	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
0380	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
0390	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
03A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
03B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
03C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
03D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
03E0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
03F0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
0400	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0410	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0420	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0430	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0440	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0450	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0460	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0470	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0480	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0490	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
04A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
04B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Code Watch Window

Note: Code Watch Window is available only when mikroICD is selected as a debugger.

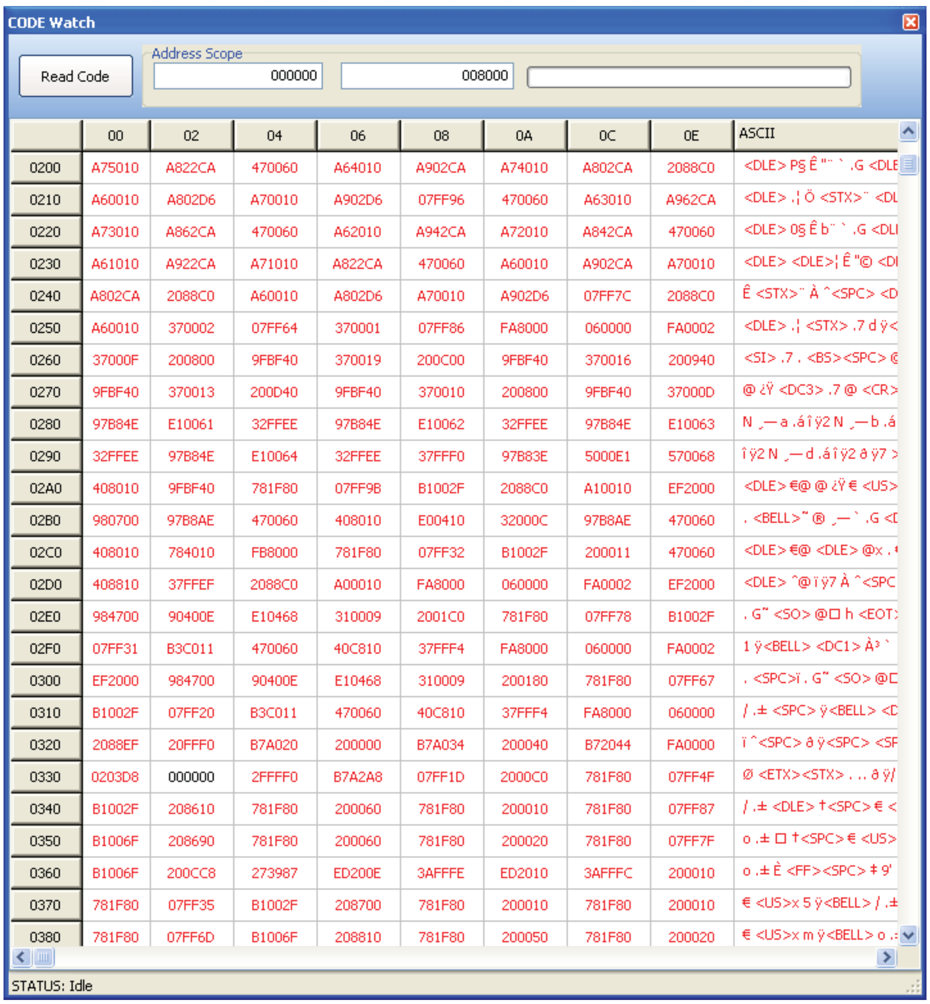
To show the Code Watch Window, select **Debug Windows › Code** from the **View** drop-down menu.

The Code Watch Window shows code (hex format) written into the MCU.

There is one action button concerning the Code Watch Window:

- Reads code from the MCU and loads it up into the Code Window. Code reading is resources consuming operation so the user should wait until the reading is over.

Also, you can set an address scope in which hex code will be read.



	00	02	04	06	08	0A	0C	0E	ASCII
0200	A75010	A822CA	470060	A64010	A902CA	A74010	A802CA	2088C0	<DLE> P§ Ê "¨ ` .G <DLE
0210	A60010	A802D6	A70010	A902D6	07FF96	470060	A63010	A962CA	<DLE> .¦ Ö <STX>¨ <DL
0220	A73010	A862CA	470060	A62010	A942CA	A72010	A842CA	470060	<DLE> 0§ Ê b¨ ` .G <DL
0230	A61010	A922CA	A71010	A822CA	470060	A60010	A902CA	A70010	<DLE> <DLE>¦ Ê "© <D
0240	A802CA	2088C0	A60010	A802D6	A70010	A902D6	07FF7C	2088C0	Ê <STX>¨ À ˆ<SPC> <D
0250	A60010	370002	07FF64	370001	07FF86	FA8000	060000	FA0002	<DLE> .¦ <STX> .7 d ÿ<
0260	37000F	200800	9FBF40	370019	200C00	9FBF40	370016	200940	<SI> .7 . <BS><SPC> @
0270	9FBF40	370013	200D40	9FBF40	370010	200800	9FBF40	37000D	@ ¿Ÿ <DC3> .7 @ <CR>
0280	97B84E	E10061	32FFEE	97B84E	E10062	32FFEE	97B84E	E10063	N ¸— a .á î ÿ2 N ¸— b .á
0290	32FFEE	97B84E	E10064	32FFEE	37FFF0	97B83E	5000E1	570068	î ÿ2 N ¸— d .á î ÿ2 ð ÿ7 >
02A0	408010	9FBF40	781F80	07FF9B	B1002F	2088C0	A10010	EF2000	<DLE> €@ @ ¿Ÿ € <US>
02B0	980700	97B8AE	470060	408010	E00410	32000C	97B8AE	470060	. <BELL>˜ ® ¸— ` .G <D
02C0	408010	784010	FB8000	781F80	07FF32	B1002F	200011	470060	<DLE> €@ <DLE> @x .
02D0	408810	37FFEF	2088C0	A00010	FA8000	060000	FA0002	EF2000	<DLE> ˆ@ ï ÿ7 À ˆ<SPC
02E0	984700	90400E	E10468	310009	2001C0	781F80	07FF78	B1002F	. G˜ <SO> @□ h <EOT
02F0	07FF31	B3C011	470060	40C810	37FFF4	FA8000	060000	FA0002	1 ÿ<BELL> <DC1> À³ `
0300	EF2000	984700	90400E	E10468	310009	200180	781F80	07FF67	. <SPC>ï . G˜ <SO> @□
0310	B1002F	07FF20	B3C011	470060	40C810	37FFF4	FA8000	060000	/ .± <SPC> ÿ<BELL> <D
0320	2088EF	20FFF0	B7A020	200000	B7A034	200040	B72044	FA0000	ï ˆ<SPC> ð ÿ<SPC> <SP
0330	0203D8	000000	2FFFF0	B7A2A8	07FF1D	2000C0	781F80	07FF4F	Ø <ETX><STX> . . . ð ÿ/
0340	B1002F	208610	781F80	200060	781F80	200010	781F80	07FF87	/ .± <DLE> †<SPC> € <
0350	B1006F	208690	781F80	200060	781F80	200020	781F80	07FF7F	o .± □ †<SPC> € <US>
0360	B1006F	200CC8	273987	ED200E	3AFFFE	ED2010	3AFFFC	200010	o .± È <FF><SPC> ‡ 9'
0370	781F80	07FF35	B1002F	208700	781F80	200010	781F80	200010	€ <US>x 5 ÿ<BELL> / .±
0380	781F80	07FF6D	B1006F	208810	781F80	200050	781F80	200020	€ <US>x m ÿ<BELL> o .±

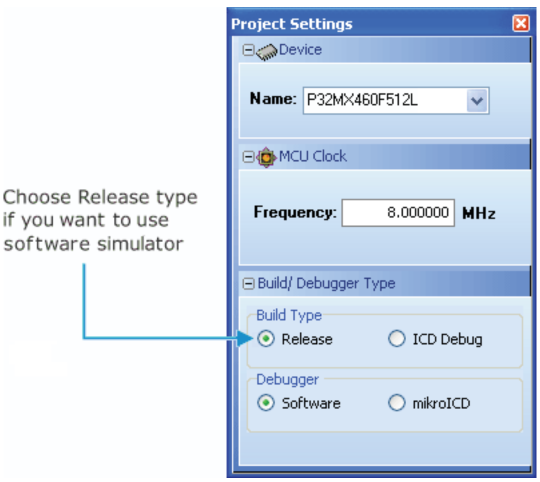
CHAPTER 5

Software Simulator Overview

Software Simulator

The Source-level Software Simulator is an integral component of the mikroPascal PRO for PIC32 environment. It is designed to simulate operations of the Microchip PIC32 MCUs and assist the users in debugging code written for these devices.

Upon completion of writing your program, choose **Release** build Type in the Project Settings window:



After you have successfuly compiled your project, you can run the Software Simulator by selecting **Run › Start Debugger** from the drop-down menu, or by clicking the Start Debugger Icon from the Debugger Toolbar.

Starting the Software Simulator makes more options available: Step Into, Step Over, Step Out, Run to Cursor, etc. Line that is to be executed is color highlighted (blue by default).

Note: The Software Simulator simulates the program flow and execution of instruction lines, but it cannot fully emulate PIC32 device behavior, i.e. it doesn't update timers, interrupt flags, etc.

Related topics: Software Simulator Debug Windows, Software Simulator Debugger Options

Software Simulator Debug Windows

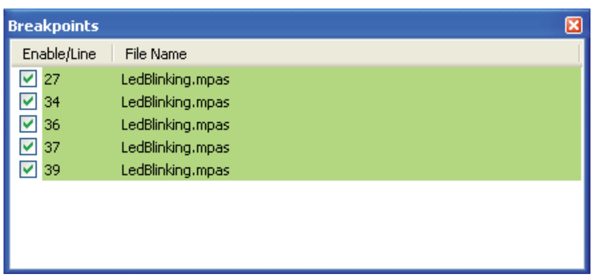
Debug Windows

This section provides an overview of available Debug Windows in mikroPascal PRO for PIC32:

- Breakpoints Window
- Watch Values Window
- RAM Window
- Stopwatch Window
- EEPROM Watch Window
- Code Watch Window

Breakpoints Window

The Breakpoints window manages the list of currently set breakpoints in the project. Doubleclicking the desired breakpoint will cause cursor to navigate to the corresponding location in source code.
In situations when multiple breakpoints are used within the code, it is sometimes handy to enable/disable certain breakpoints. To do this, just check/uncheck the desired breakpoint using the checkbox in front of the breakpoint's name.



Watch Values Window

Watch Values Window is the main Debugger window which allows you to monitor program execution. To show the Watch Values Window, select **Debug Windows › Watch** from the **View** drop-down menu.

The Watch Values Window displays variables and registers of the MCU, with their addresses and values. Values are updated along with the code execution. Recently changed items are coloured red.

There are two ways to add variable/register into the watch list:

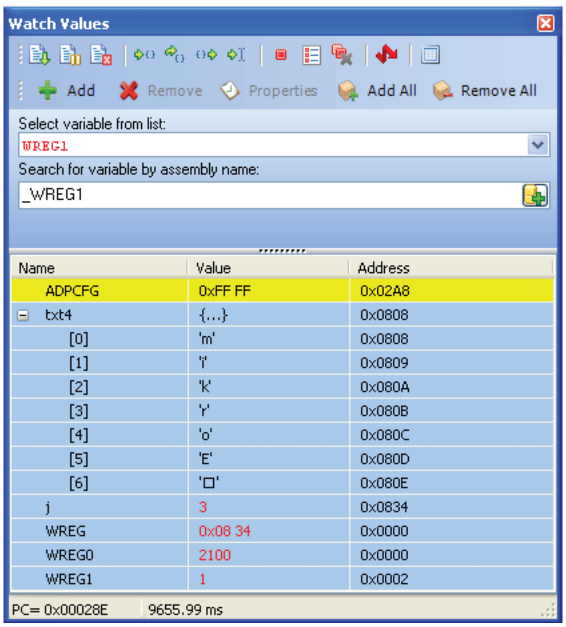
- by its real name (variable's name in program code). Just select wanted variable/register from **Select variable from list** drop-down menu and click the Add button.
- by its name ID (assembly variable name). Simply type name ID of the variable/register you want to display into **Search for variable by assemby name** box and click the Add button.

Also, it is possible to add all variables in the Watch Values Window by clicking Add All button.

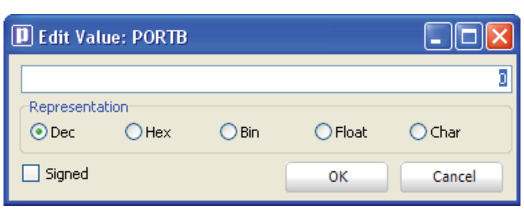
To remove a variable from the Watch Values Window, just select the variable that you want to remove and then click the Remove button, or press the Delete key.

It is possible to remove all variables from the Watch Values Window by clicking Remove All button.

You can also expand/collapse complex variables i.e. struct type variables, strings, etc, by clicking the appropriate button (+ or −) beside variable name.



Double clicking a variable or clicking the Properties button opens the Edit Value window in which you can assign a new value to the selected variable/register. Also, you can choose the format of variable/register representation between decimal, hexadecimal, binary, float or character. All representations except float are unsigned by default. For signed representation click the check box next to the **Signed** label.



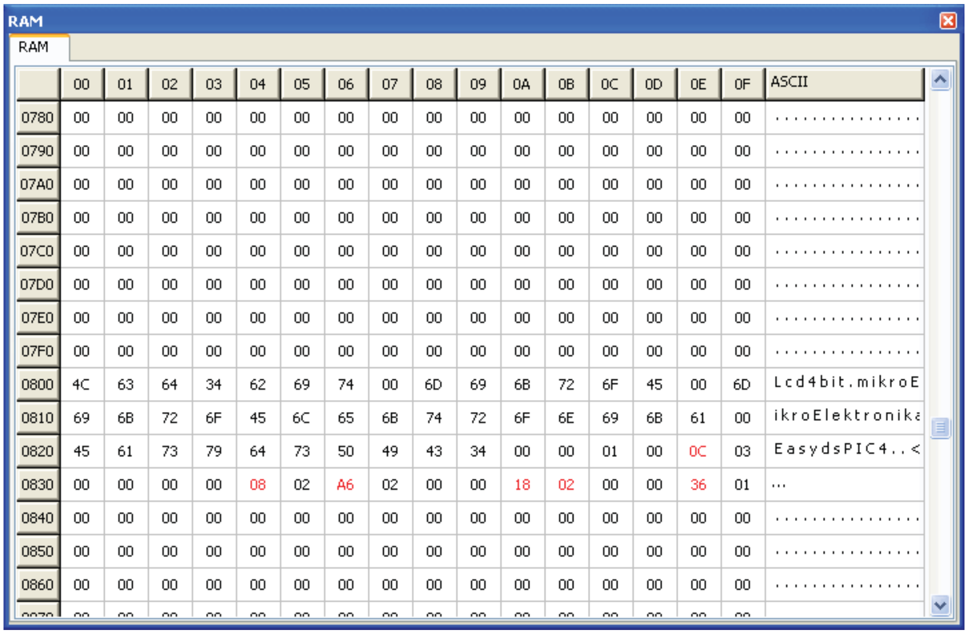
An item's value can also be changed by double clicking item's value field and typing the new value directly.

RAM Window

The RAM Window is available from the drop-down menu, **View › Debug Windows › RAM**.

The RAM Window displays the map of MCU’s RAM, with recently changed items colored red. The user can edit and change the values in the RAM window.

mikroICD Specific: RAM window content will be written to the MCU before the next instruction execution.

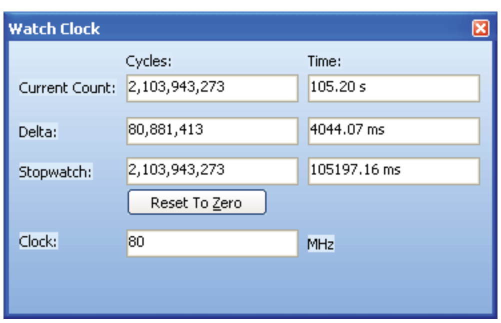


	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0780	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0790	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
07A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
07B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
07C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
07D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
07E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
07F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0800	4C	63	64	34	62	69	74	00	6D	69	6B	72	6F	45	00	6D	Lcd4bit.mikroE
0810	69	6B	72	6F	45	6C	65	6B	74	72	6F	6E	69	6B	61	00	ikroElektronika
0820	45	61	73	79	64	73	50	49	43	34	00	00	01	00	0C	03	EasydsPIC4..<
0830	00	00	00	00	08	02	A6	02	00	00	18	02	00	00	36	01	...
0840	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0850	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0860	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Stopwatch Window

The Software Simulator Stopwatch Window is available from the drop-down menu, **View › Debug Windows › Stopwatch**.

The Stopwatch Window displays a `Current Count` of cycles/time since the last Software Simulator action.
`Stopwatch` measures the execution time (number of cycles) from the moment Software Simulator has started and can be reset at any time.
`Delta` represents the number of cycles between the lines where Software Simulator action has started and ended.



Notes:

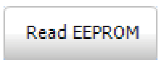
- The user can change the clock in the Stopwatch Window, which will recalculate values for the latest specified frequency.
- Changing the clock in the Stopwatch Window does not affect actual project settings – it only provides a simulation.
- Stopwatch is available only when Software Simulator is selected as a debugger.

EEPROM Watch Window

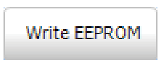
Note: EEPROM Watch Window is available only when mikroICD is selected as a debugger.

To show the EEPROM Watch Window, select **Debug Windows › EEPROM** from the **View** drop-down menu. The EEPROM Watch Window shows current content of the MCU's internal EEPROM memory.

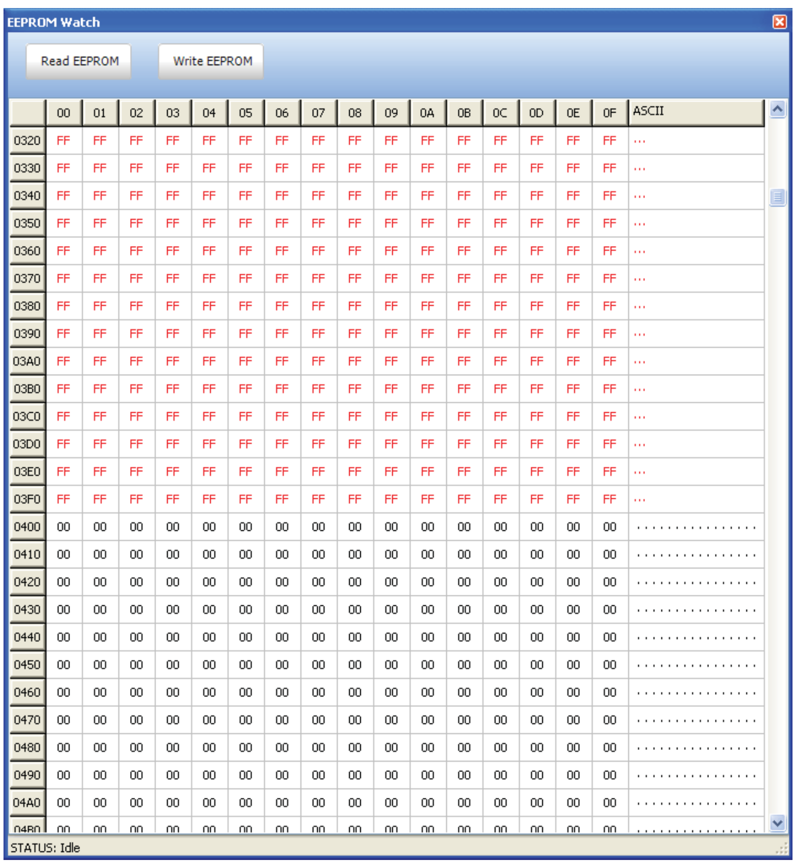
There are two action buttons concerning the EEPROM Watch Window:



- Reads data from MCU's internal EEPROM memory and loads it up into the EEPROM window.



- Writes data from the EEPROM window into MCU's internal EEPROM memory.



Code Watch Window

Note: Code Watch Window is available only when mikroICD is selected as a debugger.

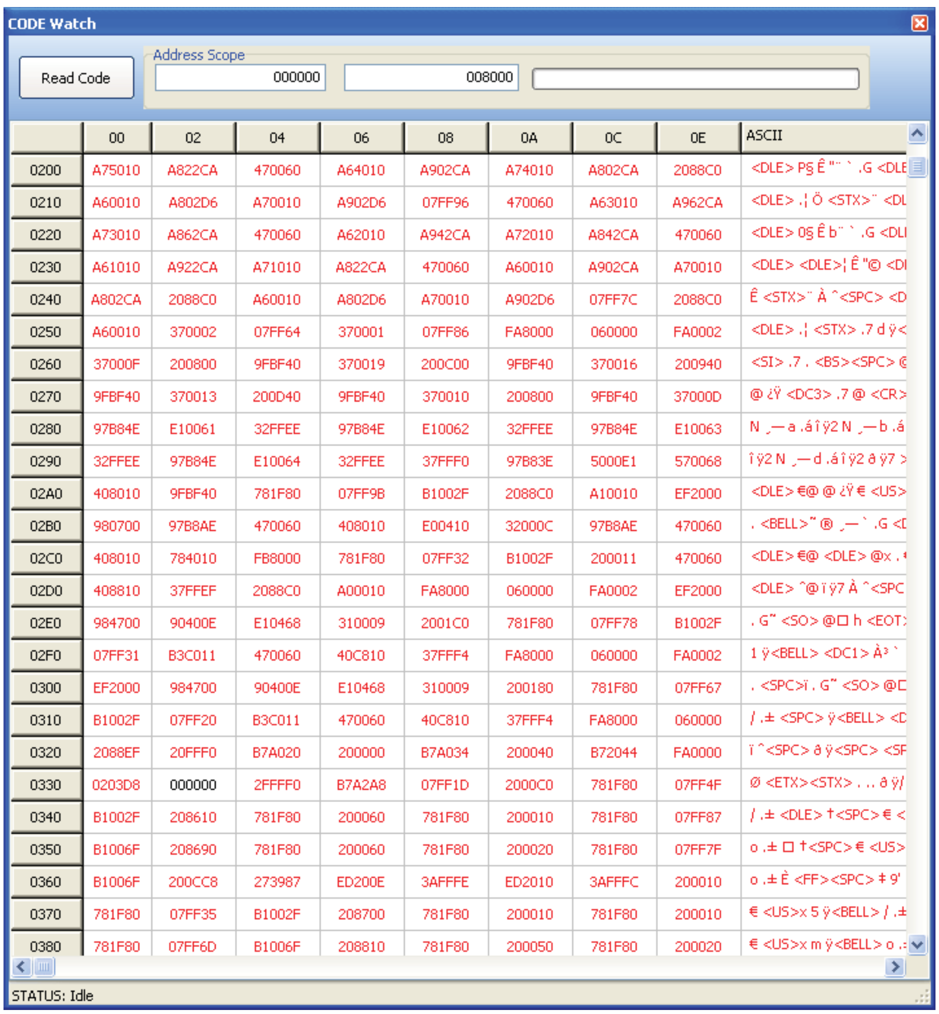
To show the Code Watch Window, select **Debug Windows › Code** from the **View** drop-down menu.

The Code Watch Window shows code (hex format) written into the MCU.

There is one action button concerning the Code Watch Window:

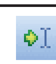
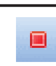
- Reads code from the MCU and loads it up into the Code Window. Code reading is resources consuming operation so the user should wait until the reading is over.

Also, you can set an address scope in which hex code will be read.



Software Simulator Debugger Options

Debugger Options

Name	Description	Function Key	Toolbar Icon
`Start Debugger`	Starts Debugger.	F9	
`Stop Debugger`	Stop Debugger.	Ctrl + F2	
`Run/Pause Debugger`	Run/Pause Debugger.	F6	
`Step Into`	Executes the current program line, then halts. If the executed program line calls another routine, the debugger steps into the routine and halts after executing the first instruction within it.	F7	
`Step Over`	Executes the current program line, then halts. If the executed program line calls another routine, the debugger will not step into it. The whole routine will be executed and the debugger halts at the first instruction following the call.	F8	
`Step Out`	Executes all remaining program lines within the subroutine. The debugger halts immediately upon exiting the subroutine.	F8	
`Run To Cursor`	Executes the program until reaching the cursor position.	Ctrl + F8	
`Toggle Breakpoint`	Toggle breakpoints option sets new breakpoints or removes those already set at the current cursor position.	F5	

Related topics: Run Menu, Debug Toolbar

CHAPTER 6

mikroPascal PRO for PIC32 Specifics

The following topics cover the specifics of mikroPascal PRO for PIC32 compiler:

- ANSI Standard Issues
- Predefined Globals and Constants
- Accessing Individual Bits
- Interrupts
- Linker Directives
- Built-in Routines
- Code Optimization

Predefined Globals and Constants

To facilitate PIC32 programming, the mikroPascal PRO for PIC32 implements a number of predefined globals and constants.

All PIC32 SFRs are implicitly declared as global variables of volatile word. These identifiers have an external linkage, and are visible in the entire project. When creating a project, the mikroPascal PRO for PIC32 will include an appropriate (*.mpas) file from defs folder, containing declarations of available SFRs and constants (such as PORTB, ADPCFG, etc). All identifiers are in upper case, identical to nomenclature in the Microchip datasheets.

For a complete set of predefined globals and constants, look for "Defs" in the mikroPascal PRO for PIC32 installation folder, or probe the Code Assistant for specific letters (Ctrl+Space in the Code Editor).

Predefined project level defines

mikroPascal PRO for PIC32 provides predefined project level defines that you can use in your project :

First one is equal to the name of selected device for the project. For example:

```
{$IFDEF P32MX460F512L}
...
{$ENDIF}
```

Second one is equal to the family name :

```
{$IFDEF PIC32}
...
{$ENDIF}
```

Related topics: Project Level Defines

Accessing Individual Bits

The mikroPascal PRO for PIC32 allows you to access individual bits of 32-bit variables. It also supports sbit and bit data types.

Lets use the Zero bit as an example. This bit is defined in the definition file of the particular MCU as :

```
const Z = 1;
var Z_bit : sbit at SR.B1;
```

To access this bit in your code by its name, you can write something like this:

```
// Clear Zero Bit
SR.Z := 0;
```

In this way, if Zero bit changes its position in the register, you are sure that the appropriate bit will be affected.
But, if Zero bit is not located in the designated register, you may get errors.

Another way of accesing bits is by using the direct member selector (`.`) with a variable, followed by a primary expression. Primary expression can be variable, constant, function call or an expression enclosed by parentheses. For individual bit access there are predefined global constants `B0, B1, … , B31,` or `0, 1, … 31,` with `31` being the most significant bit :

```
// predefined globals as bit designators
// Clear bit 0 in STATUS register
SR.B0 := 0;

// literal constant as bit designator
// Set bit 5 in STATUS register
SR.5 := 1;

// expression as bit designator
// Set bit 6 in STATUS register
i := 5;
SR.(i+1) := 1;
```

In this way, if the target bit changes its position in the register, you cannot be sure that you are invoking the appropriate bit.

When using literal constants as bit designators instead of predefined ones, make sure not to exceed the appropriate type size.

This kind of selective access is an intrinsic feature of mikroPascal PRO for PIC32 and can be used anywhere in the code. Identifiers B0–B31 are not case sensitive and have a specific namespace. You may override them with your own members B0–B31 within any given structure.

Also, you can access the desired bit by using its alias name, in this case `Z_bit` :

```
// Set Zero Bit
Z_bit := 1;
```

In this way, if the Zero bit changes its register or position in the register, you are sure that the appropriate bit will be affected.

See Predefined Globals and Constants for more information on register/bit names.

sbit type

The mikroPascal PRO for PIC32 compiler has sbit data type which provides access to bit-addressable SFRs.
You can declare a sbit varible in a unit in such way that it points to a specific bit in SFR register:

```
unit MyUnit;

var Abit: sbit; sfr; external; // Abit is precisely defined in some external file, for example in the main program unit
...
implementation
....
end.
```

In the main program you have to specify to which register this sbit points to, for example:

```
program MyProgram;
...
var Abit: sbit at PORTB.0; // this is where Abit is fully defined
...
begin
...
end.
```

In this way the variable `Abit` will actually point to PORTB.0. Please note that we used the keyword sfr for declaration of `Abit`, because we are pointing it to PORTB which is defined as a sfr variable.
In case we want to declare a bit over a variable which is not defined as `sfr`, then the keyword `sfr` is not necessary, for example:

```
unit MyUnit;

var AnotherBit: sbit; external; // Abit is precisely defined in some external file, for example in the main program unit
...
implementation
...
end.

program MyProgram;
...
var MyVar: byte;
var Abit: sbit at MyVar.0; // this is where Abit is fully defined
...
begin
...
end.
```

at keyword

You can use the keyword “at” to make an alias to a variable, for example, you can write a library without using register names, and later in the main program to define those registers, for example:

```
unit MyUnit;

var PORTAlias: byte; external; // here in the library we can use its symbolic name
...
implementation
...
end.

program MyProgram;
...
var PORTAlias: byte at PORTB; // this is where PORTAlias is fully defined
...
begin
...
end.
```

Note : Bear in mind that when using `at` operator in your code over a variable defined through a `external` modifier, appropriate memory specifer must be appended also.

bit type

The mikroPascal PRO for PIC32 compiler provides a bit data type that may be used for variable declarations. It can not be used for argument lists, and function-return values.

```
var bf : bit;    // bit variable
```

There are no pointers to bit variables:

```
var ptr : ^bit;    // invalid
```

An array of type bit is not valid:

```
var arr[5] : bit;     // invalid
```

Note :

- Bit variables can not be initialized.
- Bit variables can not be members of records.
- Bit variables do not have addresses, therefore unary operator @ (address of) is not applicable to these variables.

Related topics: Predefined globals and constants, External modifier

Interrupts

The PIC32MX generates interrupt requests in response to interrupt events from peripheral modules. The Interrupt module exists external to the CPU logic and prioritizes the interrupt events before presenting them to the CPU.
The PIC32MX Interrupts module includes the following features:

- Up to 96 interrupt sources.
- Up to 64 interrupt vectors.
- Single and Multi-Vector mode operations.
- Five external interrupts with edge polarity control.
- Interrupt proximity timer.
- Module freeze in Debug mode.
- Seven user-selectable priority levels for each vector.
- Four user-selectable subpriority levels within each priority.
- User-configurable shadow set based on priority level (this feature is not available on all devices; refer to the specific device data sheet for availability).
- Software can generate any interrupt.
- User-configurable interrupt vector table location.

ISRs are organized in IVT. ISR is defined as a standard function but with the `iv` directive afterwards which connects the function with specific interrupt vector. For more information on IVT refer to the PIC32 Family Reference Manual.

Configuring Interrupts

The PIC32MX interrupt controller can be configured to operate in one of two modes:

- **Single Vector mode** - all interrupt requests will be serviced at one vector address (mode out of reset).
- **Multi-Vector mode** - interrupt requests will be serviced at the calculated vector address.

Single Vector Mode

In this mode, the CPU always vectors to the same address. This means that only one ISR can be defined. The Single Vector mode address is calculated by using the Exception Base (EBase) address (its address default is 0x9FC01000E). The exact formula for Single Vector mode is as follows: **Single Vector Address = EBase + 0x200.**

Multi Vector Mode

In this mode, the CPU vectors to the unique address for each vector number. Each vector is located at a specific offset, with respect to a base address specified by the EBase register in the CPU.
The individual vector address offset is determined by the following equation: **EBase + (Vector_Number x Vector_Space) + 0x200.**

By default, the compiler configures interrupts in the **Multi Vector** mode, with the **EBase** address set to **0x9FC01000** and **vector spacing of 32**.
Configuring the Interrupt operating mode is performed in the Edit Project window.

Interrupt Priorities

In the Multi Vector Mode, the user is able to assign a **group priority** and **group subpriority level** to each of the interrupt vectors. The user-selectable priority levels range from 1 (the lowest priority) to 7 (the highest).
If an interrupt priority is set to zero, the interrupt vector is disabled for both interrupt and wake-up purposes. Interrupt vectors with a higher priority level preempt lower priority interrupts.

The subpriority will cause that when two interrupts with the same priority are pending, the interrupt with the highest subpriority will be handled first. The user-selectable subpriority levels range from 0 (the lowest subpriority) to 3 (the highest).

Interrupts and Register Sets

The PIC32MX family of devices employs two register sets, a **primary register set** for normal program execution and a **shadow register set** for highest priority interrupt processing.

Register Set Selection in Single Vector Mode

In Single Vector mode, you can select which register set will be used. By default, the interrupt controller will instruct the CPU to use the first register set. This can be changed later in the code.

Interrupts and Register Sets

When a priority level interrupt matches a shadow set priority, the interrupt controller instructs the CPU to use the shadow set. For all other interrupt priorities, the interrupt controller instructs the CPU to use the primary register set.

Interrupt Coding Requirements

In order to correctly utilize interrupts and correctly write the ISR code, the user will need to take care of these things:

1. Write the Interrupt Service Routine. You may use Interrupt Assistant to easily write this routine.
2. Initialize the module which will generate an interrupt.
3. Set the correct priority and subpriority for the used module according to the priorities set in the Interrupt Service Routine.
4. Enable Interrupts.

Interrupt Service Routine

Interrupt service routine is defined in this way:

```
procedure interrupt(); iv IVT_ADC; ilevel 7; ics ICS_SOFT;
begin
  // Interrupt service routine code
end;
```

where:

- **iv** - reserved word that inform the compiler that it is an interrupt service routine.
- **IVT_ADC** - appropriate Interrupt Vector.
- **ilevel** 7 - Interrupt priority level 7.
- **ics** Interrupt Context Saving; Interrupt Context Saving can be performed in several ways:

  1. `ICS_SOFT` - Context saving is carried out by the software.
  2. `ICS_SRS` - Shadow Register set is use for context saving.
  3. `ICS_OFF` - No context saving
  4. `ICS_AUTO` - Compiler chooses whether the `ICS_SOFT` or `ICS_SRS` will be used.

User can explicitly declare starting interrupt routine address using `org` directive:

```
procedure interrupt(); org 0x9D000000; iv IVT_ADC; ilevel 7; ics ICS_SOFT;
begin
  // Interrupt service routine code
end;
```

Function Calls from Interrupt

Calling functions from within the interrupt routine is possible. The compiler takes care about the registers being used, both in “interrupt” and in “main” thread, and performs “smart” context-switching between two of them, saving only the registers that have been used in both threads. It is not recommended to use a function call from interrupt. In case of doing that take care of stack depth.

Use the DisableContextSaving to instruct the compiler not to automatically perform context-switching. This means that no register will be saved/restored by the compiler on entrance/exit from interrupt service routine.
This enables the user to manually write code for saving registers upon entrance and to restore them before exit from interrupt.

Interrupt Example

Here is a simple example of handling the interrupts from `Timer1` (if no other interrupts are allowed):

```
program Timer1_interrupt;

procedure Timer1Int(); iv IVT_TIMER_1; ilevel 7; ics ICS_SRS;
begin
  T1IF_bit := 0;              // Clear T1IF
  LATB := not PORTB;          // Invert PORTB
end;

begin
  AD1PCFG := 0xFFFF;          // Initialize AN pins as digital
  TRISB   := 0;               // initialize PORTB as output
  LATB    := 0xAAAA;          // Initialize PORTB value

  TMR1 := 0;                  // reset timer value to zero
  PR1 := 65000;               // Load period register

  T1IP0_bit := 1;             // set interrupt
  T1IP1_bit := 1;             // priority
  T1IP2_bit := 1;             // to 7

  TCKPS0_bit := 1;            // Set Timer Input Clock
  TCKPS1_bit := 1;            // Prescale value to 1:256

  EnableInterrupts();

  T1IE_bit := 1;              // Enable Timer1 Interrupt
  ON__T1CON_bit := 1;         // Enable Timer1
end.
```

Linker Directives

mikroPascal PRO for PIC32 uses an internal algorithm to distribute objects within memory. If you need to have a variable, constant or a routine at the specific predefined address, use the linker directives absolute and org.

When using these directives, be sure to use them in proper memory segments, i.e. for functions it is the KSEG0 and for variables it is the KSEG1. Linker directives are used with the **virtual addresses**.

Directive absolute

Directive absolute specifies the starting address in RAM for a variable. If the variable is multi-byte, higher bytes will be stored at the consecutive locations.

Directive `absolute` is appended to declaration of a variable:

```
// Variable x will occupy 1 word (16 bits) at address 0xA0000000
var x : word; absolute 0xA0000000;

// Variable y will occupy 2 words at addresses 0xA0000000 and 0xA0000002
var y : longint; absolute 0xA0000000;
```

Be careful when using the absolute directive, as you may overlap two variables by accident. For example:

```
// Variable i will occupy 1 word at address 0xA0000002;
var  i : word; absolute 0xA0000002;

// Variable will occupy 2 words at 0xA0000000 and 0xA0000002; thus,
// changing i changes jj at the same time and vice versa
var jj : longint; absolute 0xA0000000;
```

Directive org

Directive org specifies the starting address of a constant or a routine in ROM. It is appended to the constant or a routine declaration.

To place a constant array in Flash memory, write the following :

```
// Constant array MONTHS will be placed starting from the address 0x9D000000
const  MONTHS  :  array[1..12]  of  byte  =  (31,28,31,30,31,30,31,31,30,31,30,31);  org 
0x800;
```

If you want to place simple type constant into Flash memory, instead of following declaration:

```
const SimpleConstant : byte = 0xAA; org 0x9D000000;
```

use an array consisting of single element :

```
const SimpleConstant : array[1] of byte = (0xAA); org 0x9D000000;
```

In first case, compiler will recognize your attempt, but in order to save Flash space, and boost performance, it will automatically replace all instances of this constant in code with it's literal value.
In the second case your constant will be placed in Flash in the exact location specified.

To place a routine on a specific address in Flash memory you should write the following :

```
procedure proc(par : byte); org 0x9D000000;
begin
// Procedure will start at address 0x9D000000;
...
end;
```

org directive can be used with main routine too. For example:

```
program Led_Blinking;

begin org 0x9D000000;             // main procedure starts at 0x9D000000
...
end.
```

Directive orgall

Use the orgall directive to specify the address above which all routines and constants will be placed. Example:

```
begin
  orgall(0x9D000000); // All the routines, constants in main program will be above the address 0x9D000000
  ...
end.
```

Built-in Routines

mikroPascal PRO for PIC32 compiler provides a set of useful built-in utility functions. Built-in functions do not have any special requirements. You can use them in any part of your project.

The `Delay_us` and `Delay_ms` routines are implemented as “inline”; i.e. code is generated in the place of a call, so the call doesn’t count against the nested call limit.

The `Vdelay_ms`, `Vdelay_advanced_ms`, `Delay_Cyc`, `Delay_Cyc_Long`, `Get_Fosc_kHz` and `Get_Fosc_Per_Cyc` are actual Pascal routines. Their sources can be found in the delays.mpas file located in the uses folder of the compiler.

- Lo
- Hi
- Higher
- Highest
- LoWord
- HiWord

- Inc
- Dec

- Chr
- Ord

- SetBit
- ClearBit
- TestBit

- Delay_us
- Delay_ms
- Vdelay_ms
- Vdelay_Advanced_ms
- Delay_Cyc
- Delay_Cyc_Long

- Clock_kHz
- Clock_MHz
- Get_Fosc_kHz
- Get_Fosc_Per_Cyc

- Reset
- ClrWdt

- DisableContextSaving

- SetFuncCall
- SetOrg

- GetDateTime
- GetVersion

- KVA0_TO_KVA1
- KVA1_TO_KVA0
- KVA_TO_PA
- PA_TO_KVA0
- PA_TO_KVA1
- CP0_Get
- CP0_Set
- EnableInterrupts
- DisableInterrupts

Lo

Prototype	`function Lo(number: longint): byte;`
Description	Function returns the lowest byte of `number`. Function does not interpret bit patterns of `number` – it merely returns 8 bits as found in register. This is an "inline" routine; code is generated in the place of the call, so the call doesn't count against the nested call limit.
Parameters	- `number`: input number
Returns	Lowest 8 bits (byte) of number, bits 7..0.
Requires	Arguments must be variable of scalar type (i.e. Arithmetic Types and Pointers).

Example	`d := 0x12345678;`
	`tmp := Lo(d);  // Equals 0x78`
	
	`Lo(d) := 0xAA; // d equals 0x123456AA`

Notes	None.

Hi

Prototype	`function Hi(number: longint): byte;`
Description	Function returns next to the lowest byte of `number`. Function does not interpret bit patterns of `number` – it merely returns 8 bits as found in register. This is an "inline" routine; code is generated in the place of the call, so the call doesn't count against the nested call limit.
Parameters	- `number`: input value
Returns	Returns next to the lowest byte of `number`, bits 8..15.
Requires	Arguments must be variable of scalar type (i.e. Arithmetic Types and Pointers).

Example	`d := 0x12345678;`
	`tmp := Hi(d);  // Equals 0x56`
	
	`Hi(d) := 0xAA; // d equals 0x1234AA78`

Notes	None.

Higher

Prototype	`function Higher(number: longint): byte;`
Description	Function returns next to the highest byte of number. Function does not interpret bit patterns of number – it merely returns 8 bits as found in register. This is an “inline” routine; code is generated in the place of the call, so the call doesn't count against the nested call limit.
Parameters	- `number`: input number
Returns	Returns next to the highest byte of `number`, bits 16..23.
Requires	Arguments must be variable of scalar type (i.e. Arithmetic Types and Pointers).

Example	`d := 0x12345678;`
	`tmp := Higher(d);  // Equals 0x34`
	
	`Higher(d) := 0xAA; // d equals 0x12AA5678`

Notes	None.

Highest

Prototype	`function Highest(number: longint): byte;`
Description	Function returns the highest byte of number. Function does not interpret bit patterns of number – it merely returns 8 bits as found in register. This is an “inline” routine; code is generated in the place of the call, so the call doesn't count against the nested call limit.
Parameters	- `number`: input number
Returns	Returns the highest byte of `number`, bits 24..31.
Requires	Arguments must be variable of scalar type (i.e. Arithmetic Types and Pointers).

Example	`d := 0x12345678;`
	`tmp := Highest(d);  // Equals 0x12`
	
	`Highest(d) := 0xAA; // d equals 0xAA345678`

Notes	None.

LoWord

Prototype	`function LoWord(val : longint) : word;`
Description	The function returns low word of val. The function does not interpret bit patterns of `val` – it merely returns 16 bits as found in register. Parameters : - `val`: input value
Parameters	`number`
Returns	Low word of `val`, bits `15..0`.
Requires	Nothing.

Example	`d := 0x12345678;`
	`tmp := LoWord(d);  // Equals 0x5678`
	
	`LoWord(d) := 0xAAAA; // d equals 0x1234AAAA`

Notes	None.

HiWord

Prototype	`function HiWord(val : longint) : word;`
Description	The function returns high word of `val`. The function does not interpret bit patterns of `val` – it merely returns 16 bits as found in register. Parameters : - `val`: input value
Parameters	`number`
Returns	High word of `val`, bits `31..16`.
Requires	Nothing.

Example	`d := 0x12345678;`
	`tmp := HiWord(d);  // Equals 0x1234`
	
	`HiWord(d) := 0xAAAA; // d equals 0xAAAA5678`

Notes	None.

Inc

Prototype	`procedure Inc(var par : longint);`
Description	Increases parameter par by 1.
Parameters	- `par`: value which will be incremented by 1
Returns	Nothing.
Requires	Nothing.
Example	`p := 4;` `Inc(p);  // p is now 5`
Notes	None.

Dec

Prototype	`procedure Dec(var par : longint);`
Description	Decreases parameter par by 1.
Parameters	- `par`: value which will be decremented by 1
Returns	Nothing.
Requires	Nothing.
Example	`p := 4;` `Dec(p);  // p is now 3`
Notes	None.

Chr

Prototype	`function Chr(code_ : byte) : char;`
Description	Function returns a character associated with the specified character `code_`. Numbers from 0 to 31 are the standard non-printable ASCII codes. This is an “inline” routine; the code is generated in the place of the call.
Parameters	- `code_`: input character
Returns	Nothing.
Requires	Nothing.
Example	`c := Chr(10);  // returns the linefeed character`
Notes	None.

Ord

Prototype	`function Ord(const character : char) : byte;`
Description	Function returns ASCII code of the character. This is an "inline" routine; the code is generated in the place of the call.
Parameters	- `character`: input character
Returns	ASCII code of the character.
Requires	Nothing.
Example	`c := Ord('A');  // returns 65`
Notes	None.

SetBit

Prototype	`procedure SetBit(var register_ : word; rbit : byte);`
Description	Function sets the bit `rbit` of `register_`. Parameter rbit needs to be a variable or literal with value 0..15. For more information on register identifiers see Predefined Globals and Constants . This is an "inline" routine; the code is generated in the place of the call.
Parameters	- `register_`: desired register - `rbit`: desired bit
Returns	Nothing.
Requires	Nothing.
Example	`SetBit(PORTB, 2);  // Set RB2`
Notes	None.

ClearBit

Prototype	`procedure ClearBit(var register_ : byte; rbit : byte);`
Description	Function clears the bit rbit of `register`. Parameter rbit needs to be a variable or literal with value 0..7. See Predefined globals and constants for more information on register identifiers. This is an "inline" routine; code is generated in the place of the call, so the call doesn't count against the nested call limit.
Parameters	- `register_`: desired register - `rbit`: desired bit
Returns	Nothing.
Requires	Nothing.
Example	`ClearBit(PORTC, 7);  // Clear RC7`
Notes	None.

TestBit

Prototype	`function TestBit(register_, rbit : byte) : byte;`
Description	Function tests if the bit `rbit` of `register` is set. If set, function returns 1, otherwise returns 0. Parameter `rbit` needs to be a variable or literal with value 0..7. See Predefined globals and constants for more information on register identifiers. This is an “inline” routine; code is generated in the place of the call, so the call doesn’t count against the nested call limit.
Parameters	- `register_`: desired register - `rbit`: desired bit
Returns	If the bit is set, returns 1, otherwise returns 0.
Requires	Nothing.
Example	`flag := TestBit(PORTE, 2);   // 1 if RE2 is set, otherwise 0`
Notes	None.

Delay_us

Prototype	`procedure Delay_us(Time_In_us: dword);`
Description	Creates a software delay in duration of `Time_In_us` microseconds. This is an “inline” routine; the code is generated in the place of the call, so the call doesn’t count against the nested call limit.
Parameters	- `Time_In_us`: delay time in microseconds. Valid values: constant values, range of applicable constants depends on the oscillator frequency
Returns	Nothing.
Requires	Nothing.
Example	`Delay_us(10);   // Ten microseconds pause`
Notes	None.

Delay_ms

Prototype	`procedure Delay_ms(Time_In_ms: dword);`
Description	Creates a software delay in duration of Time_In_ms milliseconds. This is an “inline” routine; the code is generated in the place of the call, so the call doesn’t count against the nested call limit.
Parameters	- `Time_In_ms`: delay time in milliseconds. Valid values: constant values, range of applicable constants depends on the oscillator frequency
Returns	Nothing.
Requires	Nothing.
Example	`Delay_ms(1000);   // One second pause`
Notes	For generating delays with variable as input parameter use the Vdelay_ms routine.

VDelay_ms

Prototype	`procedure VDelay_ms(Time_ms : word);`
Description	Creates a software delay in duration of Time_ms milliseconds. Generated delay is not as precise as the delay created by Delay_ms.
Parameters	- `Time_ms`: delay time in milliseconds
Returns	Nothing.
Requires	Nothing.

Example	`var pause : word;`
	`...`
	`VDelay_ms(pause);  // ~ one second pause`

Notes	None.

VDelay_advanced_ms

Prototype	`procedure VDelay_advanced_ms(time_ms, Current_Fosc_kHz: word);`
Description	Creates a software delay in duration of time_in_ms milliseconds (a variable), for a given oscillator frequency. Generated delay is not as precise as the delay created by Delay_ms. Note that Vdelay_ms is library function rather than a built-in routine; it is presented in this topic for the sake of convenience.
Parameters	- `time_ms`: delay time in milliseconds - `Current_Fosc_kHz`: frequency in kHz
Returns	Nothing.
Requires	Nothing.

Example	`pause := 1000;`
	`fosc := 10000;`
	
	`VDelay_advanced_ms(pause, fosc);  // Generates approximately one second pause, for a oscillator frequency of 10 MHz`

Notes	None.

Delay_Cyc

Prototype	`procedure Delay_Cyc(x: word; y: word);`
Description	Creates a delay based on MCU clock. Delay lasts for x*16384 + y MCU clock cycles.
Parameters	- x: NumberOfCycles divided by 16384 - y: remainder of the NumberOfCycles/16384 division
Returns	Nothing.
Requires	Nothing.
Example	`Delay_Cyc(1, 10);  // 1x16384 + 10 = 16394 cycles pause`
Notes	`Delay_Cyc` is a library function rather than a built-in routine; it is presented in this topic for the sake of convenience.

Delay_Cyc_Long

Prototype	`procedure Delay_Cyc_Long(CycNo : word);`
Description	Creates a delay based on MCU clock. Delay lasts for CycNo MCU clock cycles.
Parameters	- `CycNo`: number of MCU cycles
Returns	Nothing.
Requires	Nothing.
Example	`Delay_Cyc_Long(16384);  // 16384 cycles pause`
Notes	`Delay_Cyc_Long` is a library function rather than a built-in routine; it is presented in this topic for the sake of convenience.

Clock_kHz

Prototype	`function Clock_kHz() : longint;`
Description	Returns device clock in kHz, rounded to the nearest integer. This is an “inline” routine; the code is generated in the place of the call.
Parameters	None.
Returns	Device clock in kHz, rounded to the nearest integer.
Requires	Nothing.
Example	`clk := Clock_kHz();`
Notes	None.

Clock_MHz

Prototype	`function Clock_MHz() : word;`
Description	Returns device clock in MHz, rounded to the nearest integer. This is an “inline” routine; the code is generated in the place of the call.
Parameters	None.
Returns	Device clock in MHz, rounded to the nearest integer.
Requires	Nothing.
Example	`clk := Clock_MHz();`
Notes	None.

Get_Fosc_kHz

Prototype	`function Get_Fosc_kHz() : longint;`
Description	Function returns device clock in kHz, rounded to the nearest integer.
Parameters	None.
Returns	Device clock in kHz.
Requires	Nothing.
Example	`clk := Get_Fosc_kHz();`
Notes	- `Get_Fosc_kHz` is a library function rather than a built-in routine; it is presented in this topic for the sake of convenience.

Get_Fosc_Per_Cyc

Prototype	`function Get_Fosc_Per_Cyc() : word;`
Description	Function returns device's clock per cycle, rounded to the nearest integer. Note that `Get_Fosc_Per_Cyc` is library function rather than a built-in routine; it is presented in this topic for the sake of convenience.
Parameters	None.
Returns	Device clock in kHz.
Requires	Nothing.

Example	`var clk_per_cyc : word;`
	`...`
	`clk_per_cyc := Get_Fosc_Per_Cyc();`

Notes	None.

Reset

Prototype	`procedure Reset();`
Description	This procedure is equal to assembler instruction **reset**.
Parameters	None.
Returns	Nothing.
Requires	Nothing.
Example	`Reset(); // Resets the MCU`
Notes	None.

ClrWdt

Prototype	`procedure ClrWdt();`
Description	This procedure is equal to assembler instruction `clrwdt`.
Parameters	None.
Returns	Nothing.
Requires	Nothing.
Example	`ClrWdt(); // Clears WDT`
Notes	None.

DisableContextSaving

Prototype	`procedure DisableContextSaving();`
Description	Use the DisableContextSaving() to instruct the compiler not to automatically perform context-switching. This means that no register will be saved/restored by the compiler on entrance/exit from interrupt service routine. This enables the user to manually write code for saving registers upon entrance and to restore them before exit from interrupt.
Parameters	None.
Returns	Nothing.
Requires	This routine must be called from main.
Example	`DisableContextSaving(); // instruct the compiler not to automatically perform context-switching`
Notes	None.

SetFuncCall

Prototype	`procedure SetFuncCall(FuncName: string);`
Description	If the linker encounters an indirect function call (by a pointer to function), it assumes that any routine whose address was taken anywhere in the program can be called at that point if it's prototype matches the pointer declaration. Use the SetFuncCall directive within routine body to instruct the linker which routines can be called indirectly from that routine : `SetFunCCall (called_func[, ,...])` Routines specified in the `SetFunCCall` argument list will be linked if the routine containing `SetFunCCall` directive is called in the code no matter whether any of them was explicitly called or not. Thus, placing `SetFuncCall` directive in main will make compiler link specified routines always.
Parameters	- FuncName: function name
Returns	Nothing.
Requires	Nothing.

Example	`procedure first(p, q: byte);`
	`begin`
	`...`
	`  SetFuncCall(second); // let linker know that we will call the routine 'second'`
	`...`
	`end`

Notes	The `SetFuncCall` directive can help the linker to optimize function frame allocation in the compiled stack.

SetOrg

Prototype	`procedure SetOrg(RoutineName: string; address: longint);`
Description	Use the `SetOrg()`; routine to specify the starting address of a routine in ROM.
Parameters	- `RoutineName`: routine name - `address`: starting address
Returns	Nothing.
Requires	This routine must be called from main.
Example	`SetOrg(UART1_Write, 0x1234);`
Notes	None.

DoGetDateTime

Prototype	`function DoGetDateTime() : string;`
Description	Use the `GetDateTime()` to get date and time of compilation as string in your code.
Parameters	None.
Returns	String with date and time when this routine is compiled.
Requires	Nothing.
Example	`str := GetDateTime();`
Notes	None.

GetVersion

Prototype	`function GetVersion() : string;`
Description	Use the `GetVersion()`; to get the current version of compiler.
Parameters	None.
Returns	String with current compiler version.
Requires	Nothing.
Example	`str := GetVersion(); // for example, str will take the value of '8.2.1.6''`
Notes	None.

KVA0_TO_KVA1

Prototype	`function KVA0_TO_KVA1(Address: dword) : dword;`
Description	Function converts virtual address from KSEG0 to the virtual address in the KSEG1.
Parameters	Desired Virtual address in the KSEG0.
Returns	Virtual address in the KSEG1.
Requires	Nothing.
Example	`address := KVA0_TO_KVA1(0x9FC00000);`
Notes	None.

KVA1_TO_KVA0

Prototype	`function KVA1_TO_KVA0(Address: dword) : dword;`
Description	Function converts virtual address from KSEG1 to the virtual address in the KSEG0.
Parameters	Desired Virtual address in the KSEG1.
Returns	Virtual address in the KSEG0.
Requires	Nothing.
Example	`address := KVA1_TO_KVA0(0xBFC00000);`
Notes	None.

KVA_TO_PA

Prototype	`function KVA_TO_PA(Address: dword) : dword;`
Description	Function converts virtual address from any Kernel segment to the appropriate physical address.
Parameters	Desired Virtual Address.
Returns	Appropriate physical address.
Requires	Nothing.
Example	`address := KVA_TO_PA(0xBFC00000);`
Notes	None.

PA_TO_KVA0

Prototype	`function PA_TO_KVA0(Address: dword) : dword;`
Description	Function converts physical address to the virtual address in the KSEG0.
Parameters	Desired physical address.
Returns	Appropriate virtual address in the KSEG0.
Requires	Nothing.
Example	`address := PA_TO_KVA0(0x1D000000);`
Notes	None.

PA_TO_KVA1

Prototype	`function PA_TO_KVA1(Address: dword) : dword;`
Description	Function converts physical address to the virtual address in the KSEG1.
Parameters	Appropriate virtual address in the KSEG1.
Returns	Virtual address in the KSEG1.
Requires	Nothing.
Example	`address := PA_TO_KVA1(0x1D000000);`
Notes	None.

CP0_GET

Prototype	`function CP0_GET(const register: TCP0REG): dword;`
Description	Function returns the value of the coprocessor register or part of the register, based upon the argument entered.
Parameters	Parameter must be a constant from the enumerated built-in constants list, which can be found at the bottom of this page.
Returns	Value of the coprocessor register or part of the register.
Requires	Nothing.

Example	`var register_value : dword;`
	
	`register_value := CP0_GET(CP0_CONFIG);`

Notes	None.

CP0_SET

Prototype	`procedure CP0_SET(const register: TCP0REG; value: dword);`
Description	Function sets the value of the coprocessor register or part of the register, based upon the register argument.
Parameters	Function sets the value of the coprocessor register or part of the register, based upon the register argument.
Returns	Nothing.
Requires	Nothing.
Example	`CP0_SET(CP0_CONFIG, 0x1A2C0000);`
Notes	None.

EnableInterrupts

Prototype	`procedure EnableInterrupts();`
Description	Function enables interrupts.
Parameters	None.
Returns	Nothing.
Requires	Nothing.
Example	`EnableInterrupts();`
Notes	None.

DisableInterrupts

Prototype	`procedure DisableInterrupts();`
Description	Function disables interrupts.
Parameters	None.
Returns	Nothing.
Requires	Nothing.
Example	`EnableInterrupts();`
Notes	None.

Coprocessor Registers				
CP0_HWRENA	CP0_BADVADDR	CP0_COUNT	CP0_COMPARE	CP0_STATUS
CP0_INTCTL	CP0_SRSCTL	CP0_SRSMAP	CP0_CAUSE	CP0_EPC
CP0_PRID	CP0_EBASE	CP0_CONFIG	CP0_CONFIG1	CP0_CONFIG2
CP0_CONFIG3	CP0_DEBUG	CP0_TRACECONTROL	CP0_TRACECONTROL2	CP0_USERTRACEDATA
CP0_TRACEBPC	CP0_DEBUG2	CP0_DEPC	CP0_ERROREPC	CP0_DESAVE

Copressor Register Fields				
CP0_HWRENA_MASK	CP0_STATUS_IE	CP0_STATUS_EXL	CP0_STATUS_ERL	CP0_STATUS_UM
CP0_STATUS_IM0	CP0_STATUS_IM1	CP0_STATUS_IPL	CP0_STATUS_IM2	CP0_STATUS_IM3
CP0_STATUS_IM4	CP0_STATUS_IM5	CP0_STATUS_IM6	CP0_STATUS_IM7	CP0_STATUS_CEE
CP0_STATUS_NMI	_CPO_STATUS_SR	CP0_STATUS_TS	CP0_STATUS_BEV	CP0_STATUS_RE
CP0_STATUS_FR	CP0_STATUS_RP	CP0_STATUS_CU0	CP0_STATUS_CU1	CP0_STATUS_CU2
CP0_STATUS_CU3	CP0_INTCTL_VS	CP0_INTCTL_IPPCI	CP0_INTCTL_IPTI	CP0_SRSCTL_CSS
CP0_SRSCTL_PSS	CP0_SRSCTL_ESS	CP0_SRSCTL_EICSS	CP0_SRSCTL_HSS	CP0_SRSMAP_SSV0
CP0_SRSMAP_SSV1	CP0_SRSMAP_SSV2	CP0_SRSMAP_SSV3	CP0_SRSMAP_SSV4	CP0_SRSMAP_SSV5
CP0_SRSMAP_SSV6	CP0_SRSMAP_SSV7	CP0_CAUSE_EXCCODE	CP0_CAUSE_IP0	CP0_CAUSE_IP1
CP0_CAUSE_RIPL	CP0_CAUSE_IP2	CP0_CAUSE_IP3	CP0_CAUSE_IP4	CP0_CAUSE_IP5
CP0_CAUSE_IP6	CP0_CAUSE_IP7	CP0_CAUSE_WP	CP0_CAUSE_IV	CP0_CAUSE_PCI
CP0_CAUSE_DC	CP0_CAUSE_CE	CP0_CAUSE_TI	CP0_CAUSE_BD	CP0_PRID_REVISION
CP0_PRID_PATCHREV	CP0_PRID_MINORREV	CP0_PRID_MAJORREV	CP0_PRID_PROCESSORID	CP0_PRID_COMPANYID
CP0_EBASE_CPUNUM	CP0_EBASE_EBASE	CP0_CONFIG_K0	CP0_CONFIG_MT	CP0_CONFIG_AR
CP0_CONFIG_AT	CP0_CONFIG_BE	CP0_CONFIG_DS	CP0_CONFIG_MDU	CP0_CONFIG_SB
CP0_CONFIG_UDI	CP0_CONFIG_KU	CP0_CONFIG1_M	CP0_CONFIG1_FP	CP0_CONFIG1_EP
CP0_CONFIG1_CA	CP0_CONFIG1_WR	CP0_CONFIG1_PC	CP0_CONFIG1_MD	CP0_CONFIG1_C2
CP0_CONFIG1_DA	CP0_CONFIG1_DL	CP0_CONFIG1_DS	CP0_CONFIG1_IA	CP0_CONFIG1_IL
CP0_CONFIG1_IS	CP0_CONFIG1_MMUSIZE	CP0_CONFIG1_M	CP0_CONFIG2_M	CP0_CONFIG3_TL
CP0_CONFIG3_SM	CP0_CONFIG3_SP	CP0_CONFIG3_VINT	CP0_CONFIG3_VEIC	CP0_CONFIG3_ITL
CP0_CONFIG3_M	CP0_DEBUG_DSS	CP0_DEBUG_DBP	CP0_DEBUG_DDBL	CP0_DEBUG_DDBS
CP0_DEBUG_DIB	CP0_DEBUG_DINT	CP0_DEBUG_DIBIMPR	CP0_DEBUG_R	CP0_DEBUG_SST
CP0_DEBUG_NOSST	CP0_DEBUG_DEXCCODE	CP0_DEBUG_VER	CP0_DEBUG_DDBLIMPR	CP0_DEBUG_DDBSIMPR
CP0_DEBUG_IEXI	CP0_DEBUG_DBUSEP	CP0_DEBUG_CACHEEP	CP0_DEBUG_MCHECKP	CP0_DEBUG_IBUSEP
CP0_DEBUG_COUNTDM	CP0_DEBUG_HALT	CP0_DEBUG_DOZE	CP0_DEBUG_LSNM	CP0_DEBUG_NODCR
CP0_DEBUG_DM	CP0_DEBUG_DBD	CP0_TRACECONTROL_ON	CP0_TRACECONTROL_MODE	CP0_TRACECONTROL_G
CP0_TRACECONTROL_ASID	CP0_TRACECONTROL_U	CP0_TRACECONTROL_0	CP0_TRACECONTROL_K	CP0_TRACECONTROL_E
CP0_TRACECONTROL_D	CP0_TRACECONTROL_IO	CP0_TRACECONTROL_TB	CP0_TRACECONTROL_UT	CP0_TRACECONTROL_TS
CP0_TRACECONTROL2_SYP	CP0_TRACECONTROL2_TBU	CP0_TRACECONTROL2_TBI	CP0_TRACECONTROL2_VALIDMODES	CP0_USERTRACEDATA_DATA
CP0_TRACEBPC_IBPON	CP0_TRACEBPC_IE	CP0_TRACEBPC_DBPON	CP0_TRACEBPC_DE	CP0_DEBUG2_PACO
CP0_DEBUG2_TUP	CP0_DEBUG2_DQ	CP0_DEBUG2_PRM		

Code Optimization

Optimizer has been added to extend the compiler usability, cut down the amount of code generated and speed-up its execution. The main features are:

Constant folding

All expressions that can be evaluated in the compile time (i.e. constant) are being replaced by their results. (3 + 5 -> 8);

Constant propagation

When a constant value is being assigned to a certain variable, the compiler recognizes this and replaces the use of the variable by constant in the code that follows, as long as the value of a variable remains unchanged.

Copy propagation

The compiler recognizes that two variables have the same value and eliminates one of them further in the code.

Value numbering

The compiler "recognizes" if two expressions yield the same result and can therefore eliminate the entire computation for one of them.

"Dead code" ellimination

The code snippets that are not being used elsewhere in the programme do not affect the final result of the application. They are automatically removed.

Stack allocation

Temporary registers ("Stacks") are being used more rationally, allowing VERY complex expressions to be evaluated with a minimum stack consumption.

Local vars optimization

No local variables are being used if their result does not affect some of the global or volatile variables.

Better code generation and local optimization

Code generation is more consistent and more attention is payed to implement specific solutions for the code "building bricks" that further reduce output code size.

Related topics: SSA Optimization, PIC32 specifics, mikroPascal PRO for PIC32 specifics, Memory type specifiers

Single Static Assignment Optimization

Introduction

In compiler design, static single assignment form (often abbreviated as SSA form or SSA) is an intermediate representation (IR) in which every variable is assigned exactly once.

An SSA-based compiler modifies the program representation so that every time a variable is assigned in the original program, a new version of the variable is created.
A new version of the variable is distinguished (renamed) by subscripting the variable name with its version number or an index, so that every definition of each variable in a program becomes unique.
At a joining point of the control flow graph where two or more different definitions of a variable meet, a hypothetical function called a phi-function is inserted so that these multiple definitions are merged.

In mikroPascal PRO for PIC32, SSA's main goal is in allocating local variables into the RX space (instead onto the frame).
To do that, SSA has to make an alias and data flow analysis of the Control Flow Graph.

Besides these savings, there are a number of compiler optimization algorithms enhanced by the use of SSA, like :

- Constant Propagation
- Dead Code Elimination
- Global Value Numbering
- Register Allocation

Changes that SSA brings is also in the way in which routine parameters are passed. When the SSA is enabled, parameters are passed through a part of the RX space which is reserved exclusively for this purpose.
Allocating local variables and parameters in RX space has its true meaning for those architectures with hardware frame.

Enabling SSA optimization in compiler is done by checking [✓] Enable SSA optimization box from the Output Settings Menu.

Lets consider a trivial case :

```
program Example;

procedure SSA_Test(y : integer; k : integer);

begin
  if (y+k) then
    asm
      nop;
    end
end;

begin
  SSA_Test(5,5);
end.
```

With SSA enabled, procedure `SSA_Test` is consisted of 3 asm instructions :

```
;Example.mpas, 29 ::              if (y+k) then
0x9D000000    0x033A1021  ADDU    R2, R25, R26
0x9D000004    0x10400002  BEQ     R2, R0, L__SSA_Test2
0x9D000008    0x70000000  NOP
```

Without SSA enabled, procedure SSA_Test is consisted of 5 asm instructions :

```
;Example.mpas, 29 ::              if (y+k) then
0x9D000000    0x87A30002  LH      R3, 2(SP)
0x9D000004    0x87A20000  LH      R2, 0(SP)
0x9D000008    0x00431021  ADDU    R2, R2, R3
0x9D00000C    0x10400002  BEQ     R2, R0, L__SSA_Test2
0x9D000010    0x70000000  NOP
```

Proper Coding Recommendations

To get the maximum out of the SSA, user should regard the following rules during the coding process :

- Routines should not contain too many parameters (not more than 4 words).
- Don't change the value of the parameter in the function body (it is better to use a new local variable).
- If the function1 parameters are passed as function2 parameters, then parameter order should remain the same :

```
procedure f2(a: integer; b: integer;) { }

procedure f1(x: integer; y: integer;) {
  // routine call
  f2(x,y);  // x->a and y->b (1 to 1 and 2 to 2) is far more efficient than :
  f2(y,x);  // y->a and x->b (1 to 2 and 2 to 1)
}
```

- Large amount of nested loops and complex structures as its members should be avoided.
- When writing a code in assembly, keep in mind that there are registers reserved exclusively for routine parameters.
- Using `goto` and `label` statements in nested loops should be avoided.
- Obtaining address of the local variable with the global pointer and using it to alter the variable's address should be avoided.

Notes :

- `emcl` files compiled with or without SSA enabled are fully compatible and can be used and mixed without any restrictions, except function pointers.
- All function prototypes and function pointers have to be built using the same optimizer because of different calling conventions in different optimizers. In SSA, function parameters are passed via working registers, and without SSA they end up on the function frame.
- This means that you cannot have a function implementation which is optimized using SSA optimizer, and to call this function via function pointer in another module which is optimized using NON-SSA.
  When using pointers to functions, compiler must know exactly how to pass function parameters and how to execute function call.

Asm code and SSA optimization

If converting code from an earlier version of the compiler, which consists of mixed asm code with the Pascal code, keep in mind that the generated code can substantially differ when SSA optimization option is enabled or disabled.
This is due to the fact that SSA optimization uses certain working registers to store routine parameters (W10-W13), rather than storing them onto the function frame.

Because of this, user must be very careful when writing asm code as existing values in the working registers used by SSA optimization can be overwritten.
To avoid this, it is recommended that user includes desired asm code in a separate routine.

Debugging Notes

SSA also influences the code debugging in such a way that the local variables will be available in the Watch Window only in those parts of the procedure where they have useful value (eg. on entering the procedure, variable isn't available until its definition).
Variables can be allocated in one part of the procedure in register W4, and in another part of the procedure in register W2, if the optimizer estimates that it is better that way. That means that the local variable has no static address.

Warning Messages Enhancement

Besides the smaller code, SSA also deals with the intensive code analysis, which in turn has the consequence in enhancing the warning messages.
For example, compiler will warn the user that the uninitialized variable is used:

```
void main() {
  int y;

  if (y)         // Variable y might not have been initialized
    PORTD = 0;
}
```

Related topics: Code Optimization, PIC32 Specifics, mikroPascal PRO for PIC32 specifics, Memory type specifiers

CHAPTER 7

PIC32 Specifics

In order to get the most from the mikroPascal PRO for PIC32 compiler, the user should be familiar with certain aspects of PIC32 MCU. This knowledge is not essential, but it can provide a better understanding of the PIC32's capabilities and limitations, and their impact on the code writing as well.

Types Efficiency

First of all, the user should know that PIC32's ALU, which performs arithmetic operations, is optimized for working with 32-bit types. Also, it performs hardware multiplication and division on the integer level,
so the floating multiplication and division is slower and consumes more memory comparing it to the integer.
The PIC32 supports 64-bit data types, but they are less efficient. They provide higher precision, but lack the code size and the execution.

Nested Calls Limitations

There are no Nested Calls Limitations, except by RAM size. A Nested call represents a function call within the function body, either to itself (recursive calls) or to another function.

Recursive calls, as a form of cross-calling, are supported by mikroPascal PRO for PIC32, but they should be used very carefully. Also calling functions from interrupt is allowed.
Calling function from both interrupt and main thread is allowed. Be careful because this programming technique may cause unpredictable results if common resources are used in both main and interrupt.

Variable, constant and routine alignment

Simple type variables whose size exceeds 2 byte (`int, long, float, double, long double`) and those exceeding 4 bytes are always set to alignment 4.
Routines are always set to aligment 4.

Boot Start-up Initialization

Upon reset, MCU positions at the address 0xBFC00000 on which the `BootStartUp` function is located.

BootStartUp function configures:

- CP0 (coprocessor registers),
- SFR registers associated with the interrupt,
- Stack pointer (R29) and global pointer (R1).

By default the MCU is configured as follows:

- Cache Enabled,
- Prefetch enabled (for executable code and constants),
- Flash waitstates set for specified oscillator frequency,
- Executable code allocated in the KSEG0,
- Data allocated in the KSEG1,

After this start-up function has been executed, MCU will jump into the main routine. Interupts will be set accordingly to the settings in the Edit Project.

PIC32 Memory Organization

The PIC32MX microcontrollers provide 4 GB of unified virtual memory address space. All memory regions, including program memory, data memory, SFRs and Configuration registers reside in this address space at their respective unique addresses.
The program and data memories can be optionally partitioned into user and kernel memories. In addition, the data memory can be made executable, allowing the PIC32MX to execute from data memory.

Key features of PIC32MX memory organization include the following:

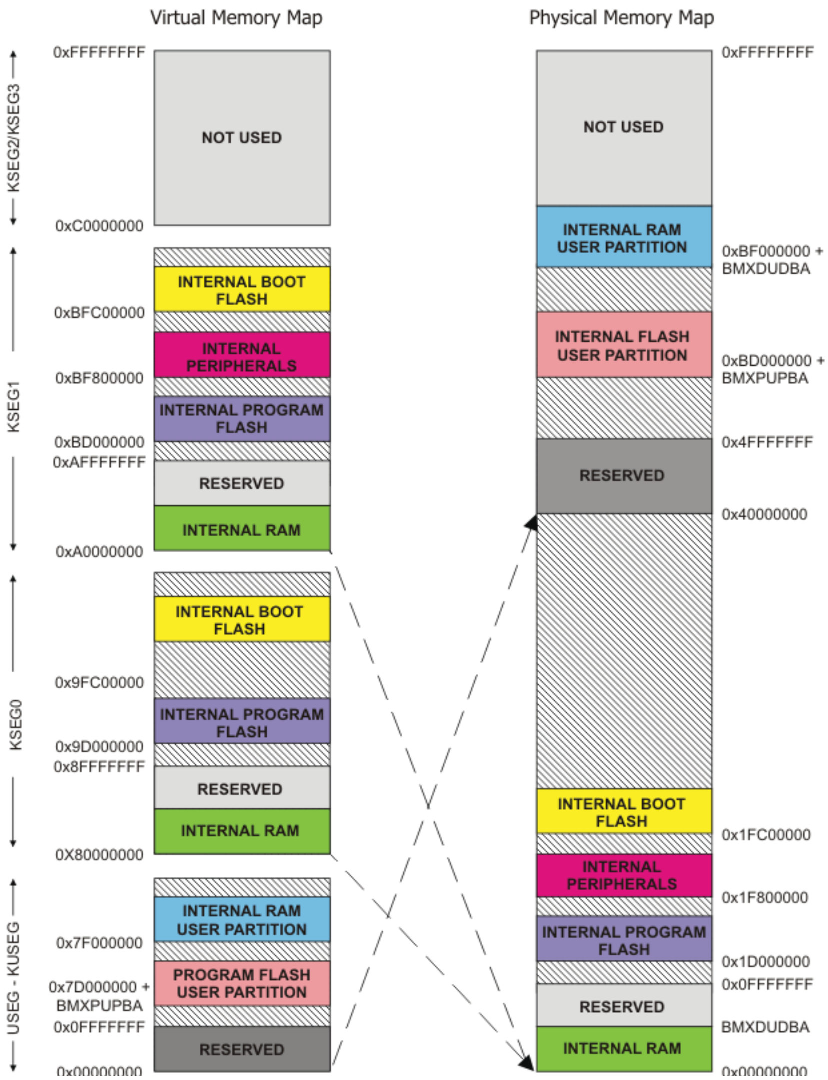
- 32-bit native data width
- Separate User and Kernel mode address spaces.
- Flexible program Flash memory partitioning.
- Flexible data RAM partitioning for data and program space.
- Separate boot Flash memory for protected code.
- Robust bus-exception handling to intercept runaway code.
- Simple memory mapping with Fixed Mapping Translation (FMT) unit.
- Cacheable and non-cacheable address regions.

PIC32MX Memory Layout

The PIC32MX microcontrollers implement two address spaces: **virtual** and **physical**.

All hardware resources, such as program memory, data memory and peripherals, are located at their respective physical addresses. Peripherals, such as DMA and Flash controllers, use physical addresses and access memory independently of the CPU.
Virtual addresses are exclusively used by the CPU to fetch and execute instructions. Virtual address space can be thought as CPU's logical view and use of its physical resources.

A graphical representation of the PIC32MX virtual and physical memory is shown in picture below:



PIC32 Memory Map

As it can be seem, the entire 4 GB virtual address space is divided into two primary regions: **User** and **Kernel** space. The lower 2 GB of space called **USEG/KUSEG**, and the upper 2 GB are divided into **KSEG0, KSEG1, KSEG2** and **KSEG3**.

Virtual vs Physical Addresses

The PIC32MX's CPU uses virtual addresses to address the peripherals, which means that to access the PIC32MX's peripherals we (and the CPU) must be operating within the virtual boundaries of KSEG1.
The PIC32MX's CPU also uses virtual addressing to fetch and execute program memory instructions.

If you look closely, you'll see that the physical address region between the **INTERNAL RAM** (at physical address 0x00000000) and the **INTERNAL BOOT FLASH** (beginning at physical address 0x1FC00000)
is matched up with the virtual memory schemes of KSEG0 and KSEG1.

The PIC32MX CPU maps the virtual areas of KSEG0 and KSEG1 against the same physical memory area beginning at physical address 0x00000000.
Because both the KSEG0 and KSEG1 virtual segments point to the same physical memory area, the PIC32MX CPU can execute instructions from either the KSEG0 or KSEG1 virtual memory segment, depending on the cacheable status of the application (KSEG0 and USEG-KSEG are cacheable while KSEG1 is not cacheable).

Related topics: Accessing individual bits, SFRs, Memory type specifiers

Memory Type Specifiers

The mikroPascal PRO for PIC32 supports usage of all memory areas.
Each variable may be explicitly assigned to a specific memory space by including a memory type specifier in the declaration, or implicitly assigned.

The following memory type specifiers can be used:

- `code`
- `data`
- `rx` (reserved for compiler purposes only)
- `sfr`

code

Description	The **code** memory type may be used for allocating constants in program memory.
Example	`// puts txt in program memory` `const txt = 'ENTER PARAMETER:'; code;`

data

Description	This memory specifier is used when storing variable to the Data RAM.
Example	`// puts data_buffer in data ram` `var data_buffer : char; data;`

rx

Description	This memory specifier allows variable to be stored in the working registers space (reserved for compiler purposes only).
Example	`// puts y in Rx space` `var y : char; rx;`

sfr

Description	This memory specifier allows user to access special function registers. It also instructs compiler to maintain same identifier in source and assembly.
Example	`var y : char; sfr; // puts y in SFR space`

Note : If none of the memory specifiers are used when declaring a variable, data specifier will be set as default by the compiler.

Related topics: PIC32 Memory Organization, Accessing individual bits, SFRs, Constants, Functions

Read Modify Write Problem

The Microchip microcontrollers use a sequence known as **Read-Modify-Write** (RMW) when changing an output state (1 or 0) on a pin. This can cause unexpected behavior under certain circumstances.

When your program changes the state on a specific pin, for example RB0 in PORTB, the microcontroller first **READs** all 8 bits of the PORTB register which represents the states of all 8 pins in PORTB (RB7-RB0).
The microcontroller then stores this data in the MCU. The bit associated with RB that you've commanded to **MODIFY** is changed, and then the microcontroller **WRITEs** all 8 bits (RB7-RB0) back to the PORTB register.

During the first reading of the PORT register, you will be reading the actual state of the physical pin.
The problem arises when an output pin is loaded in such a way that its logic state is affected by the load. Instances of such loads are LEDs without current-limiting resistors or loads with high capacitance or inductance.

For example, if a capacitor is attached between pin and ground, it will take a short while to charge when the pin is set to 1.
On the other hand, if the capacitor is discharged, it acts like a short circuit, forcing the pin to '0' state, and, therefore, a read of the PORT register will return 0, even though we wrote a 1 to it.

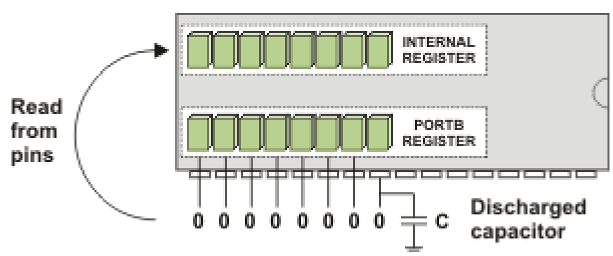
Lets analyze the following example:

```
PORTB.B0 = 1;
PORTB.B1 = 1;
```

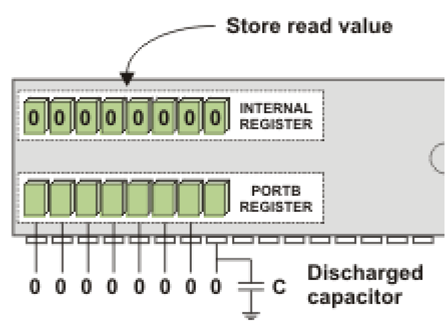
Assume that the PORTB is initially set to zero, and that all pins are set to output. Let's say we connect a discharged capacitor to RB0 pin.

The first line, `PORTB.B0 = 1`; will be decoded like in this way:

READ PORTB is **read**:

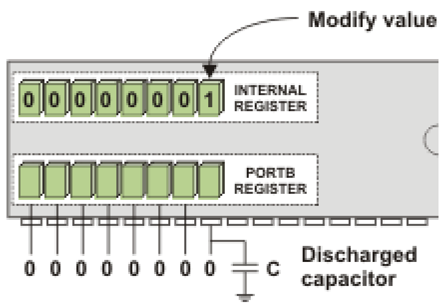


STORE Data is stored inside a temporary internal register in the MCU:

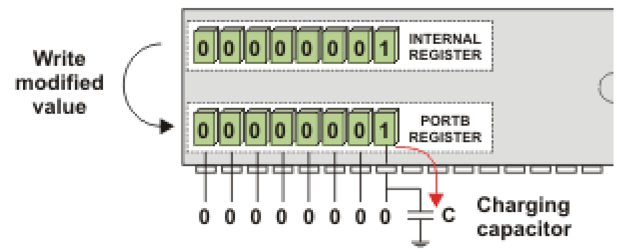


Actual voltage levels on MCU pins are relevant.

MODIFY Data is **modified** to set the RB0 bit:

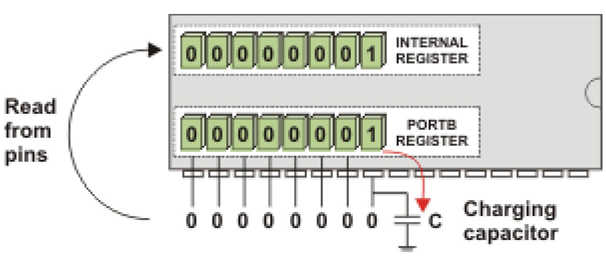


WRITE PORTB is **written** with the modified data. The output driver for RB0 turns on, and the capacitor starts to charge:

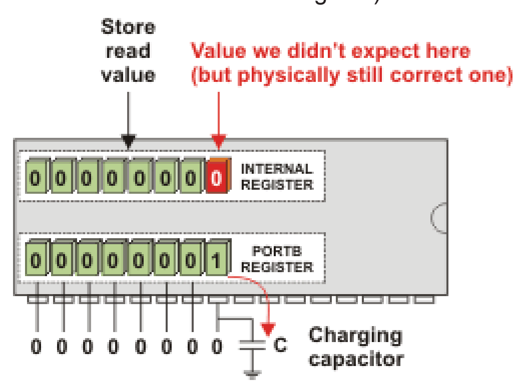


The second line, PORTB.B1 = 1; will be decoded like in this way:

READ PORTB is **read**:

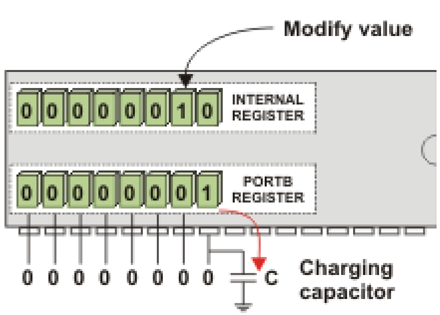


STORE Because the capacitor is still charging, the voltage at RB0 is still low and reads as a ‘0’ (since we are reading from the pins directly, not from the PORTB register) :

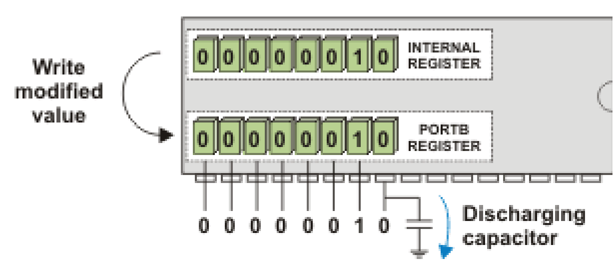


Actual voltage levels on MCU pins are relevant.

MODIFY Data is **modified** to set the bit:



WRITE PORTB is **written** with the new data. The output driver for RB1 turns on, **but the driver for RB0 turns back off:**



To correct the problem in the code, insert a delay after each `PORTB.Bx = 1` line, or modify the entire PORTB register in a single line `PORTB = 0b00000011`.

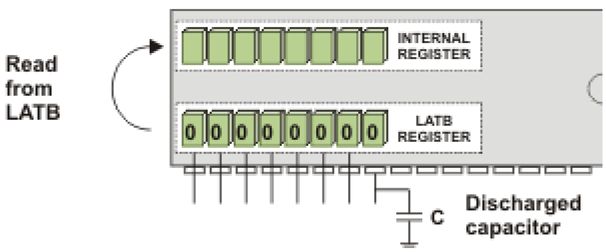
This problem can be avoided by using LATx register when writing to ports, rather than using PORTx registers. Writing to a LATx register is equivalent to writing to a PORTx register, **but readings from LATx registers return the data value held in the port latch, regardless of the state of the actual pin**.

For example, lets analyze the following example:

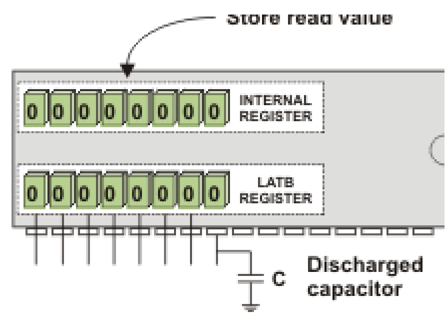
```
LATB.B0 = 1;
LATB.B1 = 1;
```

The first line, `LATB.B0 = 1`; will be decoded like in this way:

READ LATB is **read**:

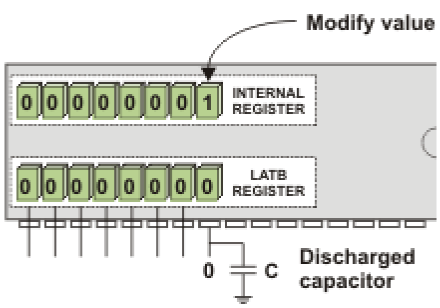


STORE Data is stored inside a temporary internal register in the MCU:

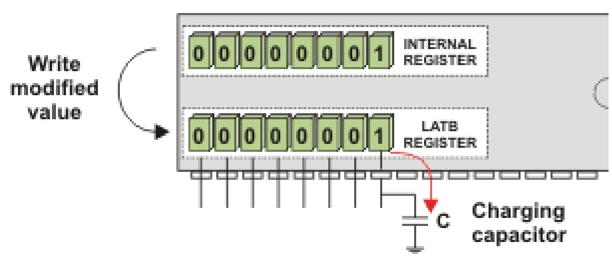


Actual voltage levels on MCU pins are no longer relevant when using LATx for output

MODIFY Data is **modified** to set the RB0 bit:

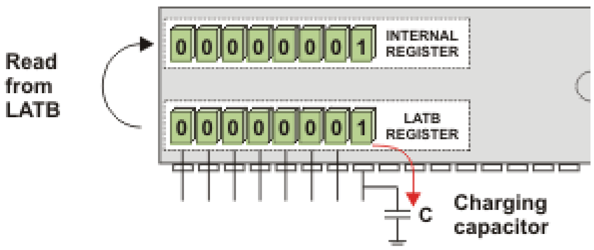


WRITE LATB is **written** with the modified data. The output driver for RB0 turns on, and the capacitor starts to charge:

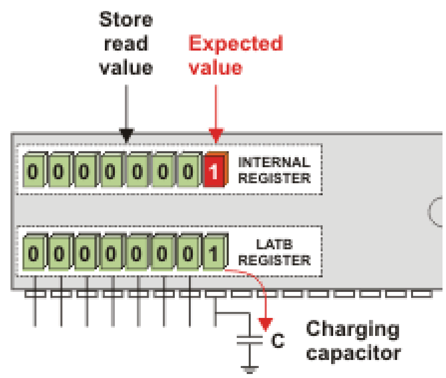


The second line, LATB.B1 = 1; will be decoded like in this way:

READ LATB is **read**:

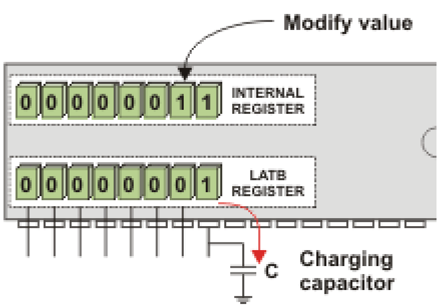


STORE Since the voltage levels on MCU pins are no longer relevant, we get the expected value:

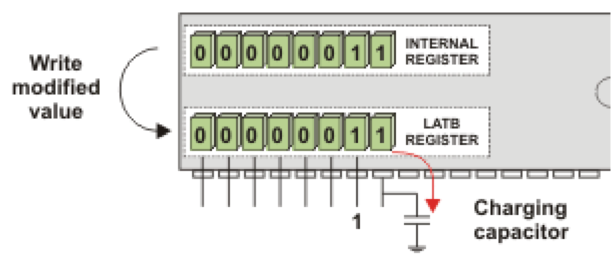


Actual voltage levels on MCU pins are no longer relevant when using LATx for output

MODIFY Data is **modified** to set the bit:



WRITE LATB is **written** with the new data. The output driver for RB1 turns on, and the output driver for RB0 remains turned on:



When to use LATx instead of PORTx

Depending on your hardware, one may experience unpredictable behavior when using PORTx bits for driving output. Displays (GLCD, LCD), chip select pins in SPI interfaces and other cases when you need fast and reliable output, **LATx should be used instead of PORTx.**

CHAPTER 8

mikroPascal PRO for PIC32 Language Reference

- Lexical Elements
  - Whitespace
  - Comments
  - Tokens
    - Literals
    - Keywords
    - Identifiers
    - Punctuators
- Program Organization
  - Program Organization
  - Scope and Visibility
  - Units
- Variables
- Constants
- Labels
- Functions and Procedures
  - Functions
  - Procedures
- Types
  - Simple Types
  - Arrays
  - Strings
  - Pointers

Lexical Elements Overview

The following topics provide a formal definition of the mikroPascal PRO for PIC32 lexical elements. They describe different categories of word-like units (tokens) recognized by the language.

In the tokenizing phase of compilation, the source code file is parsed (i.e. broken down) into tokens and whitespace. The tokens in mikroPascal PRO for PIC32 are derived from a series of operations performed on your programs by the compiler.

Whitespace

Whitespace is a collective name given to spaces (blanks), horizontal and vertical tabs, newline characters and comments. Whitespace can serve to indicate where tokens start and end, but beyond this function, any surplus whitespace is discarded.

For example, the two sequences

```
var i : char;
    j : word;
```

and

```
var
i : char;

    j : word;
```

are lexically equivalent and parse identically to give nine tokens:

```
var
i
:
char
;
j
:
word
;
```

Newline Character

Newline character (CR/LF) is not a whitespace in BASIC, and serves as a statement terminator/separator. In mikroPascal PRO for PIC32, however, you may use newline to break long statements into several lines. Parser will first try to get the longest possible expression (across lines if necessary), and then check for statement terminators.

Whitespace in Strings

The ASCII characters representing whitespace can occur within string literals, in which case they are protected from the normal parsing process (they remain a part of the string). For example,

```
some_string := 'mikro foo';
```

parses into four tokens, including a single string literal token:

```
some_string
:=
'mikro foo'
;
```

Comments

Comments are pieces of a text used to annotate a program, and are technically another form of whitespace. Comments are for the programmer's use only. They are stripped from the source text before parsing.

There are two ways to create comments in mikroPascal. You can use multi-line comments which are enclosed with braces or (* and *):

```
{ All text between left and right brace
  constitutes a comment. May span multiple lines. }

(* Comment can be
   written in this way too. *)
```

or single-line comments:

```
// Any text between a double-slash and the end of the
// line constitutes a comment spanning one line only.
```

Nested comments

mikroPascal PRO for PIC32 doesn't allow nested comments. The attempt to nest a comment like this

```
{ i { identifier } : word; }
```

fails, because the scope of the first open brace "{" ends at the first closed brace "}". This gives us

```
: word; }
```

which would generate a syntax error.

Tokens

Token is the smallest element of a mikroPascal PRO for PIC32 program, meaningful to the compiler. The parser separates tokens from the input stream by creating the longest token possible using the input characters in a left–to–right scan.

mikroPascal PRO for PIC32 recognizes the following kinds of tokens:

- keywords
- identifiers
- constants
- operators
- punctuators (also known as separators)

Token Extraction Example

Here is an example of token extraction. Take a look at the following example code sequence:

```
end_flag := 0;
```

First, note that `end_flag` would be parsed as a single identifier, rather than as the keyword `end` followed by the identifier `_flag`.

The compiler would parse it as the following four tokens:

```
end_flag   // variable identifier
:=         // assignment operator
0          // literal
;          // statement terminator
```

Note that `:=` parses as one token (the longest token possible), not as token `:` followed by token `=`.

Literals

Literals are tokens representing fixed numeric or character values.

The data type of a constant is deduced by the compiler using such clues as numeric value and format used in the source code.

Integer Literals

Integral values can be represented in decimal, hexadecimal or binary notation.

In decimal notation, numerals are represented as a sequence of digits (without commas, spaces or dots), with optional prefix + or - operator to indicate the sign. Values default to positive (`6258` is equivalent to `+6258`).

The dollar-sign prefix (`$`) or the prefix `0x` indicates a hexadecimal numeral (for example, `$8F` or `0x8F`).

The percent-sign prefix (`%`) indicates a binary numeral (for example, `%01010000`).

Here are some examples:

```
11      // decimal literal
$11     // hex literal, equals decimal 17
0x11    // hex literal, equals decimal 17
%11     // binary literal, equals decimal 3
```

The allowed range of values is imposed by the largest data type in mikroPascal PRO for PIC32 – `longint`. Compiler will report an error if the literal exceeds `2147483647` (`$7FFFFFFF`).

Floating Point Literals

A floating-point value consists of:

- Decimal integer
- Decimal point
- Decimal fraction
- `e` or `E` and a signed integer exponent (optional)

You can omit either the decimal integer or decimal fraction (but not both).

Negative floating constants are taken as positive constants with the unary operator minus (-) prefixed.

mikroPascal PRO for PIC32 limits floating-point constants to range ±1.17549435082 * 10-38 .. ±6.80564774407 * 1038.

```
0.        // = 0.0
-1.23     // = -1.23
23.45e6   // = 23.45 * 10^6
2e-5      // = 2.0 * 10^-5
3E+10     // = 3.0 * 10^10
.09E34    // = 0.09 * 10^34
```

Character Literals

Character literal is one character from the extended ASCII character set, enclosed with apostrophes.

Character literal can be assigned to variables of the `byte` and `char` type (variable of `byte` will be assigned the ASCII value of the character). Also, you can assign character literal to a string variable.

Note : Quotes (`""`) have no special meaning in mikroPascal PRO for PIC32.

String Literals

String literal is a sequence of characters from the extended ASCII character set, written in one line and enclosed with apostrophes. Whitespace is preserved in string literals, i.e. parser does not “go into” strings but treats them as single tokens.

Length of string literal is a number of characters it consists of. String is stored internally as the given sequence of characters plus a final null character. This null character is introduced to terminate the string, it does not count against the string’s total length.

String literal with nothing in between the apostrophes (null string) is stored as a single null character.

You can assign string literal to a string variable or to an array of `char`.

Here are several string literals:

```
'Hello world!'           // message, 12 chars long
'Temperature is stable' // message, 21 chars long
'  '                     // two spaces, 2 chars long
'C'                      // letter, 1 char long
''                       // null string, 0 chars long
```

The apostrophe itself cannot be a part of the string literal, i.e. there is no escape sequence. You can use the built-in function Chr to print an apostrophe: `Chr(39)`. Also, see String Splicing.

Keywords

Keywords are words reserved for special purposes and must not be used as normal identifier names.

Beside standard Pascal keywords, all relevant SFRs are defined as global variables and represent reserved words that cannot be redefined (for example: `W0`, `TMR1`, `T1CON`, etc). Probe the Code Assistant for specific letters (Ctrl+Space in Editor) or refer to Predefined Globals and Constants.

Here is the alphabetical listing of keywords in mikroPascal PRO for PIC32:

```
- absolute
- abstract
- and
- array
- as
- asm
- assembler
- at
- automated
- bdata
- begin
- bit
- case
- cdecl
- class
- code
- compact
- const
- constructor
- contains
- data
- default
- deprecated
- destructor
- dispid
- dispinterface
- div
- dma
- do
- downto
- dynamic
- end
- except
- export
- exports
- external
- far
- file
- final
- finalization
- finally
- for
- forward
- goto
- helper
- idata
- if
- ilevel
- implementation
- implements
- in
- index
- inherited
- initialization
- inline
- interface
- io
- is
- label
- library
- message
- mod
- name
- near
- nil
- nodefault
- not
- object
- of
- on
- operator
- or
- org
- out
- overload
- override
- package
- packed
- pascal
- pdata
- platform
- private
- procedure
- program
- property
- protected
- public
- published
- raise
- read
- readonly
- record
- register
- reintroduce
- repeat
- requires
- rx
- safecall
- sbit
- sealed
- set
- sfr
- shl
- shr
- small
- stdcall
- stored
- string
- threadvar
- to
- try
- type
- unit
- until
- uses
- var
- virtual
- volatile
- while
- with
- write
- writeonly
- xdata
- xor
- ydata
```

Also, mikroPascal PRO for PIC32 includes a number of predefined identifiers used in libraries. You can replace them by your own definitions, if you plan to develop your own libraries. For more information, see mikroPascal PRO for PIC32 Libraries.

Identifiers

Identifiers are arbitrary names of any length given to functions, variables, symbolic constants, user-defined data types and labels. All these program elements will be referred to as objects throughout the help (don't get confused about the meaning of object in object-oriented programming).

Identifiers can contain the letters a to z and A to Z, underscore character "_", and digits from 0 to 9. The only restriction is that the first character must be a letter or an underscore.

Case Sensitivity

mikroPascal PRO for PIC32 is not case sensitive, so `Sum`, `sum`, and `suM` are equivalent identifiers.

Uniqueness and Scope

Although identifier names are arbitrary (according to the stated rules), if the same name is used for more than one identifier within the same scope then error arises. Duplicated names are illegal within same scope. For more information, refer to Scope and Visibility.

Identifier Examples

Here are some valid identifiers:

```
temperature_V1
Pressure
no_hit
dat2string
SUM3
_vtext
```

… and here are some invalid identifiers:

```
7temp       // NO -- cannot begin with a numeral
%higher     // NO -- cannot contain special characters
xor         // NO -- cannot match reserved word
j23.07.04   // NO -- cannot contain special characters (dot)
```

Punctuators

The mikroPascal PRO for PIC32 punctuators (also known as separators) are:

[ ] – Brackets
( ) – Parentheses
, – Comma
; – Semicolon
: – Colon
. – Dot

Brackets

Brackets [ ] indicate single and multidimensional array subscripts:

```
var alphabet : array[1..30] of byte;
// ...
alphabet[3] := 'c';
```

For more information, refer to Arrays.

Parentheses

Parentheses ( ) are used to group expressions, isolate conditional expressions and indicate function calls and function declarations:

```
d := c * (a + b);           // Override normal precedence
if (d = z) then ...         // Useful with conditional statements
func();                     // Function call, no arguments
function func2(n : word);   // Function declaration with parameters
```

For more information, refer to Operators Precedence and Associativity, Expressions and Functions and Procedures.

Comma

Comma (`,`) separates the arguments in function calls:

```
LCD_Out(1, 1, txt);
```

Furthermore, the comma separates identifiers in declarations:

```
var i, j, k : byte;
```

The comma also separates elements of array in initialization lists:

```
const MONTHS : array[1..12] of byte = (31,28,31,30,31,30,31,31,30,31,30,31);
```

Semicolon

Semicolon (`;`) is a statement terminator. Every statement in Pascal must be terminated with a semicolon. The exceptions are: the last (outer most) `end` statement in the program which is terminated with a dot and the last statement before `end` which doesn't need to be terminated with a semicolon.

For more information, see Statements.

Colon

Colon (`:`) is used in declarations to separate identifier list from type identifier. For example:

```
var
  i, j : byte;
  k    : word;
```

In the program, use the colon to indicate a labeled statement:

```
start:  nop;
  ...
goto start;
```

For more information, refer to Labels.

Dot

Dot (`.`) indicates an access to a field of a record. For example:

```
person.surname := 'Smith';
```

For more information, refer to Records.

Dot is a necessary part of floating point literals. Also, dot can be used for accessing individual bits of registers in mikroPascal.

Program Organization

mikroPascal PRO for PIC32 imposes strict program organization. Below you can find models for writing legible and organized source files. For more information on file inclusion and scope, refer to Units and Scope and Visibility.

Organization of Main Unit

Basically, the main source file has two sections: declaration and program body. Declarations should be in their proper place in the code, organized in an orderly manner. Otherwise, the compiler may not be able to comprehend the program correctly.

When writing code, follow the model presented below. The main unit should look like this:

```
program { program name }
uses { include other units }

//********************************************************
//* Declarations (globals):
//********************************************************

{ constants declarations }
const ...

{ types declarations }
type ...

{ variables declarations }
var Name[, Name2...] : [^]type; [absolute 0x123;] [external;] [volatile;] [register;]
[sfr;]

{ labels declarations }
label ...

{ procedures declarations }
procedure procedure_name(parameter_list);
  { local declarations }
  begin
    ...
  end;

{ functions declarations }
function function_name(parameter_list) : return_type;
  { local declarations }
  begin
    ...
  end;

//********************************************************
//* Program body:
//********************************************************

begin
  { write your code here }
end.
```

Organization of Other Units

Units other than main start with the keyword unit. Implementation section starts with the keyword implementation. Follow the model presented below:

```
unit { unit name }
uses { include other units }

//********************************************************
//* Interface (globals):
//********************************************************

{ constants declarations }
const ...

{ types declarations }
type ...

{ variables declarations }
var Name[, Name2...] : [^]type; [absolute 0x123;] [external;] [volatile;] [register;]
[sfr;]

{ procedures prototypes }
procedure procedure_name([var] [const] ParamName : [^]type; [var] [const] ParamName2,
ParamName3 : [^]type);

{ functions prototypes }
function function_name([var] [const] ParamName : [^]type; [var] [const] ParamName2,
ParamName3 : [^]type) : [^]type;

//********************************************************
//* Implementation:
//********************************************************

implementation

{ constants declarations }
const ...

{ types declarations }
type ...

{ variables declarations }
var Name[, Name2...] : [^]type; [absolute 0x123;] [external;] [volatile;] [register;]
[sfr;]

{ labels declarations }
label ...

{  procedures declarations }
procedure procedure_name([var] [const] ParamName : [^]type; [var] [const] ParamName2,
ParamName3 : [^]type); [ilevel 0x123;] [overload;] [forward;]
  {  local declarations }
  begin
    ...
  end;
```

```
{ functions declarations }
function function_name([var] [const] ParamName : [^]type; [var] [const] ParamName2,
ParamName3 : [^]type) : [^]type; [ilevel 0x123;] [overload;] [forward;]
  { local declarations }
  begin
    ...
  end;

end.
```

Note :

- Constants, types and variables used in the implementation section are inaccessible to other units. This feature is not applied to the procedures and functions in the current version, but it will be added to the future ones.
- Functions and procedures must have the same declarations in the interface and implementation section. Otherwise, compiler will report an error.

Scope and Visibility

Scope

The scope of an identifier is a part of the program in which the identifier can be used to access its object. There are different categories of scope, which depends on how and where identifiers are declared:

Place of declaration	Scope
Identifier is declared in the declaration of a program, function, or procedure	Scope extends from the point where it is declared to the end of the current block, including all blocks enclosed within that scope. Identifiers in the outermost scope (file scope) of the main unit are referred to as globals, while other identifiers are locals.
Identifier is declared in the interface section of a unit	Scope extends the interface section of a unit from the point where it is declared to the end of the unit, and to any other unit or program that uses that unit.
Identifier is declared in the implementation section of a unit, but not within the block of any function or procedure	Scope extends from the point where it is declared to the end of the unit. The identifier is available to any function or procedure in the unit.

Visibility

The visibility of an identifier is that region of the program source code from which legal access to the identifier's associated object can be made.

Scope and visibility usually coincide, though there are circumstances under which an object becomes temporarily hidden by the appearance of a duplicate identifier, i.e. the object still exists but the original identifier cannot be used to access it until the scope of the duplicate identifier is ended.

Technically, visibility cannot exceed scope, but scope can exceed visibility

Name Spaces

Name space is a scope within which an identifier must be unique. The mikroPascal PRO for PIC32 uses two distinct categories of identifiers:

1. Global variables are visible throughout the whole unit, from the place of declaration. Also. they can be seen in other units, if they are declared above the Implementation section.
2. Local variables, parameters, types, function results - must be unique within the block in which they are declared.

For example:

```
var level : byte;

procedure control(sens : byte);
  var location : byte;
  begin
    location := 1;
    sens := location;
    level := 123;
  end;

procedure temperature;
  begin
    location := 0; // ILLEGAL
    sens := 23;    // ILLEGAL: redefinition of sens
    level := 95;
  end;
```

Units

In mikroPascal PRO for PIC32, each project consists of a single project file and one or more unit files. Project file, with extension `.mpp32` contains information about the project, while unit files, with extension .mpas, contain the actual source code.

Units allow you to:

- break large programs into encapsulated parts that can be edited separately,
- create libraries that can be used in different projects,
- distribute libraries to other developers without disclosing the source code.

Each unit is stored in its own file and compiled separately. Compiled units are linked to create an application. In order to build a project, the compiler needs either a source file or a compiled unit file (`.emcl` file) for each unit.

Uses Clause

mikroPascal PRO for PIC32 includes units by means of the `uses` clause. It consists of the reserved word uses, followed by one or more comma-delimited unit names, followed by a semicolon. Extension of the file should not be included. There can be at most one uses clause in each source file, and it must appear immediately after the program (or unit) name.

Here's an example:

```
uses utils, strings, Unit2, MyUnit;
```

For the given unit name, the compiler will check for the presence of `.emcl` and `.mpas` files, in order specified by the search paths.

- If both `.mpas` and `.emcl` files are found, the compiler will check their dates and include the newer one in the project. If the `.mpas` file is newer than `.emcl`, a new library will be written over the old one;
- If only `.mpas` file is found, the compiler will create the `.emcl` file and include it in the project;
- If only `.emcl` file is present, i.e. no source code is available, the compiler will include it as it is found;
- If none found, the compiler will issue a “File not found” warning.

Main Unit

Every project in mikroPascal PRO for PIC32 requires a single main unit file. The main unit file is identified by the keyword `program` at the beginning; it instructs the compiler where to “start”.

After you have successfully created an empty project with the Project Wizard, the Code Editor will display a new main unit. It contains the bare-bones of the Pascal program:

```
program MyProject;

{ main procedure }
begin
  { Place program code here }
end.
```

Nothing should precede the keyword program except comments. After the program name, you can optionally place the uses clause.

Place all global declarations (constants, variables, types, labels, routines) before the keyword begin.

Other Units

Units other than main start with the keyword unit. Newly created blank unit contains the bare-bones:

```
unit MyUnit;

implementation

end.
```

Other than comments, nothing should precede the keyword `unit`. After the unit name, you can optionally place the `uses` clause.

Interface Section

Part of the unit above the keyword `implementation` is referred to as interface section. Here, you can place global declarations (constants, variables, labels and types) for the project.

You do not define routines in the interface section. Instead, state the prototypes of routines (from implementation section) that you want to be visible outside the unit. Prototypes must match the declarations exactly.

Implementation Section

Implementation section hides all irrelevant innards from other units, allowing encapsulation of code.

Everything declared below the keyword `implementation` is private, i.e. has its scope limited to the file. When you declare an identifier in the implementation section of a unit, you cannot use it outside the unit, but you can use it in any block or routine defined within the unit.

By placing the prototype in the interface section of the unit (above the `implementation`) you can make the routine public, i.e. visible outside of unit. Prototypes must match the declarations exactly.

Variables

Variable is an object whose value can be changed during the runtime. Every variable is declared under unique name which must be a valid identifier. This name is used for accessing the memory location occupied by a variable.

Variables are declared in the declaration part of the file or routine — each variable needs to be declared before being used. Global variables (those that do not belong to any enclosing block) are declared below the uses statement, above the keyword begin.

Specifying a data type for each variable is mandatory. Syntax for variable declaration is:

```
var identifier_list : type;
```

Here, `identifier_list` is a comma-delimited list of valid identifiers and type can be any data type.

For more details refer to Types and Types Conversions. For more information on variables' scope refer to the chapter Scope and Visibility.

Pascal allows shortened syntax with only one keyword var followed by multiple variable declarations. For example:

```
var i, j, k : byte;
    counter, temp : word;
    samples : array[100] of word;
```

External Modifier

Use the `external` modifier to indicate that the actual place and initial value of the variable, function or procedure body, is defined in a separate source code unit.

For example, lets create a project which will calculate circle area and will have function and procedure definition in two different units, and a call to these routines in the third, separate unit.
So, the project will be consisted of the main unit, `Main_Unit.mpas` and `First_Unit.mpas` and `Second_Unit.mpas` units.

In the `Main_Unit` we will define routine called `r_squared` (calculates radius squared). Also, both units must be included in the Main_Unit :

```
program Main_Unit;

uses First_Unit, Second_Unit;  // Include both used units

function r_squared(r : real) : real;  // Definition of the r_squared routine
begin
  result := r*r;
end;

begin
  CircleArea();  // CircleArea routine call
end.
```

In the First_Unit we will define and declare routine called pi_r_squared (calculates pi multiplied by the radius squared):

```
unit First_Unit;

procedure pi_r_squared(rr : real);  // Declaration of the pi_r_squared routine

implementation

procedure pi_r_squared(rr : real);  // Definition of the pi_r_squared routine
var res : real;

begin
  res := rr*3.14;
end;

end.
```

In the Second_Unit we will make a call to the routines defined externally (`r_squared` and `pi_r_squared`). First of all, we must declare their prototypes followed with a external modifier. Then, we can proceed to the routine call :

```
unit Second_Unit;

procedure CircleArea();
function r_squared(r : real) : real; external;  // Declaration of the r_squared routine (defined in Main_Unit) followed with a external modifier
procedure pi_r_squared(rr : real); external;     // Declaration of the pi_r_squared routine (defined in First_Unit) followed with a external modifier

implementation

procedure CircleArea();  // Definition of the CircleArea routine
var res : real;

begin
   res := r_squared(5);    // r_squared routine call
   pi_r_squared(res);      // pi_r_squared routine call
end;

end.
```

Variables and PIC32

Every declared variable consumes part of RAM memory. Data type of variable determines not only the allowed range of values, but also the space a variable occupies in RAM memory. Bear in mind that operations using different types of variables take different time to be completed. mikroPascal PRO for PIC32 recycles local variable memory space – local variables declared in different functions and procedures share the same memory space, if possible.

There is no need to declare SFRs explicitly, as mikroPascal PRO for PIC32 automatically declares relevant registers as global variables of `volatile word` see SFR for details.

Constants

Constant is a data whose value cannot be changed during the runtime. Using a constant in a program consumes no RAM memory. Constants can be used in any expression, but cannot be assigned a new value.

Constants are declared in the declaration part of a program or routine. You can declare any number of constants after the keyword `const`:

```
const constant_name [: type] = value;
```

Every constant is declared under unique `constant_name` which must be a valid identifier. It is a tradition to write constant names in uppercase. Constant requires you to specify `value`, which is a literal appropriate for the given type. `type` is optional and in the absence of `type`, the compiler assumes the “smallest” of all types that can accommodate `value`.

Note : You cannot omit `type` when declaring a constant array.

Pascal allows shorthand syntax with only one keyword const followed by multiple constant declarations. Here’s an example:

```
const
  MAX : longint = 10000;
  MIN = 1000;       // compiler will assume word type
  SWITCH = 'n';     // compiler will assume char type
  MSG = 'Hello';    // compiler will assume string type
  MONTHS : array[1..12] of byte = (31,28,31,30,31,30,31,31,30,31,30,31);
```

Labels

Labels serve as targets for goto statements. Mark the desired statement with a label and colon like this:

```
label_identifier : statement
```

Before marking a statement, you must declare a label. Labels are declared in declaration part of unit or routine, similar to variables and constants. Declare labels using the keyword `label`:

```
label label1, ..., labeln;
```

Name of the label needs to be a valid identifier. The label declaration, marked statement, and `goto` statement must belong to the same block. Hence it is not possible to jump into or out of a procedure or function. Do not mark more than one statement in a block with the same label.

Here is an example of an infinite loop that calls the `Beep` procedure repeatedly:

```
label loop;
...
loop:
  Beep;
  goto loop;
```

Note : Label should be followed by end of line (CR) otherwise compiler will report an error.

```
label loop;
...
loop: Beep; // compiler will report an error
loop: // compiler will report an error
```

Functions and Procedures

Functions and procedures, collectively referred to as routines, are subprograms (self-contained statement blocks) which perform a certain task based on a number of input parameters. When executed, a function returns a value while procedure does not.

Functions

A function is declared like this:

```
function function_name(parameter_list) : return_type;
  { local declarations }
begin
  { function body }
end;
```

`function_name` represents a function's name and can be any valid identifier. `return_type` is a type of return value and can be any simple type or complex type. Within parentheses, `parameter_list` is a formal parameter list very similar to variable declaration. In Pascal, parameters are always passed to a function by the value. To pass an argument by address, add the keyword var ahead of identifier.

`Local declarations` are optional declarations of variables and/or constants, local for the given function. `Function body` is a sequence of statements to be executed upon calling the function.

Calling a function

A function is called by its name, with actual arguments placed in the same sequence as their matching formal parameters. The compiler is able to coerce mismatching arguments to the proper type according to implicit conversion rules. Upon a function call, all formal parameters are created as local objects initialized by values of actual arguments. Upon return from a function, a temporary object is created in the place of the call and it is initialized by the value of the function result. This means that function call as an operand in complex expression is treated as the function result.

In standard Pascal, a `function_name` is automatically created local variable that can be used for returning a value of a function. mikroPascal PRO for PIC32 also allows you to use the automatically created local variable result to assign the return value of a function if you find function name to be too ponderous. If the return value of a function is not defined the compiler will report an error.

Function calls are considered to be primary expressions and can be used in situations where expression is expected. A function call can also be a self-contained statement and in that case the return value is discarded.

Example

Here's a simple function which calculates x^n based on input parameters `x` and `n` (`n > 0`):

```
function power(x, n : byte) : longint;
var i : byte;
begin
  i := 0; result := 1;
  if n > 0 then
    for i := 1 to n do result := result*x;
end;
```

Now we could call it to calculate, say, 312:

```
tmp := power(3, 12);
```

Procedures

Procedure is declared like this:

```
procedure procedure_name(parameter_list);
  { local declarations }
begin
  { procedure body }
end;
```

`procedure_name` represents a procedure's name and can be any valid identifier. Within parentheses, `parameter_list` is a formal parameter list very similar to variable declaration. In Pascal, parameters are always passed to a procedure by the value — to pass an argument by address, add the keyword var ahead of identifier.

`Local declarations` are optional declaration of variables and/or constants, local for the given procedure. `Procedure body` is a sequence of statements to be executed upon calling the procedure.

Calling a procedure

A procedure is called by its name, with actual arguments placed in the same sequence as their matching formal parameters. The compiler is able to coerce mismatching arguments to the proper type according to implicit conversion rules. Upon procedure call, all formal parameters are created as local objects initialized by the values of actual arguments.

Procedure call is a self-contained statement.

Example

Here's an example procedure which transforms its input time parameters, preparing them for output on Lcd:

```
procedure time_prep(var sec, min, hr : byte);
begin
  sec  := ((sec and $F0) shr 4)*10 + (sec and $0F);
  min  := ((min and $F0) shr 4)*10 + (min and $0F);
  hr   := ((hr  and $F0) shr 4)*10 + (hr  and $0F);
end;
```

A function can return a complex type. Follow the example bellow to learn how to declare and use a function which returns a complex type.

Example:

This example shows how to declare a function which returns a complex type.

```
program Example;

type TCircle = record  // Record
    CenterX, CenterY: word;
    Radius: byte;
end;

var MyCircle: TCircle; // Global variable

function DefineCircle(x, y: word; r: byte): TCircle; // DefineCircle function returns a
Record

begin
  result.CenterX := x;
  result.CenterY := y;
  result.Radius  := r;
end;

begin
  MyCircle := DefineCircle(100, 200, 30);                    // Get a Record via function
call
  MyCircle.CenterX := DefineCircle(100, 200, 30).CenterX + 20; // Access a Record field
via function call
  //                 |-----------------------|   |-----|
  //                    |                          |
  //                 Function returns TCircle     Access to one field of TCircle
end.
```

Forward declaration

A function can be declared without having it followed by it's implementation, by having it followed by the forward procedure. The effective implementation of that function must follow later in the unit. The function can be used after a forward declaration as if it had been implemented already. The following is an example of a forward declaration:

```
program Volume;

var Volume : word;

function First(a, b : word) : word; forward;

function Second(c : word) : word;
var tmp : word;
begin
  tmp := First(2, 3);
  result := tmp * c;
end;
```

```
function First(a, b : word) : word;
begin
  result := a * b;
end;

begin
  Volume := Second(4);
end.
```

Functions reentrancy

Functions reentrancy is allowed. Remember that the PIC32 have memory limitations that can vary between MCUs.

Types

Pascal is strictly typed language, which means that every variable and constant need to have a strictly defined type, known at the time of compilation.

The type serves:

- to determine the correct memory allocation required,
- to interpret the bit patterns found in the object during subsequent accesses,
- in many type-checking situations, to ensure that illegal assignments are trapped.

mikroPascal PRO for PIC32 supports many standard (predefined) and user-defined data types, including signed and unsigned integers of various sizes, arrays, strings, pointers and records.

Type Categories

Types can be divided into:

- simple types
- arrays
- strings
- pointers
- records

Simple Types

Simple types represent types that cannot be divided into more basic elements and are the model for representing elementary data on machine level. Basic memory unit in mikroPascal PRO for PIC32 has 32 bits.

Here is an overview of simple types in mikroPascal PRO for PIC32:

Type	Size	Range
`bit`	1–bit	0 or 1
`sbit`	1–bit	0 or 1
`byte, char`	8–bit	0 .. 255
`short`	8–bit	-127 .. 128
`word`	16–bit	0 .. 65535
`integer`	16–bit	-32768 .. 32767
`dword`	32–bit	0 .. 4294967295
`longint`	32–bit	-2147483648 .. 2147483647
`real`	32–bit	±1.17549435082 * 10-38 .. ±6.80564774407 * 1038
`int64`	64–bit	-9223372036854775808 .. 9223372036854775807
`uint64`	64–bit	0 .. 18446744073709551615

You can assign signed to unsigned or vice versa only using the explicit conversion. Refer to Types Conversions for more information.

Derived Types

The derived types are also known as structured types. They are used as elements in creating more complex user-defined types.

The derived types include:

- arrays
- pointers
- records

Arrays

An array represents an indexed collection of elements of the same type (called the base type). Because each element has a unique index, arrays, unlike sets, can meaningfully contain the same value more than once.

Array Declaration

Array types are denoted by constructions in the following form:

```
array[index_start .. index_end] of type
```

Each of the elements of an array is numbered from `index_start` through `index_end`. The specifier `index_start` can be omitted along with dots, in which case it defaults to zero.

Every element of an array is of type and can be accessed by specifying array name followed by element's index within brackets.

Here are a few examples of array declaration:

```
var
  weekdays : array[1..7] of byte;
  samples  : array[50] of word;

begin
  // Now we can access elements of array variables, for example:
  samples[0] := 1;
  if samples[37] = 0 then ...
```

Constant Arrays

Constant array is initialized by assigning it a comma-delimited sequence of values within parentheses. For example:

```
// Declare a constant array which holds number of days in each month:
const MONTHS : array[1..12] of byte = (31,28,31,30,31,30,31,31,30,31,30,31);
```

The number of assigned values must not exceed the specified length. The opposite is possible, when the trailing "excess" elements are assigned zeroes.

For more information on arrays of `char`, refer to Strings.

Multi-dimensional Arrays

Multidimensional arrays are constructed by declaring arrays of array type. These arrays are stored in memory in such way that the right most subscript changes fastest, i.e. arrays are stored "in rows". Here is a sample 2-dimensional array:

```
m : array[5] of array[10] of byte;   // 2-dimensional array of size 5x10
```

A variable m is an array of 5 elements, which in turn are arrays of 10 byte each. Thus, we have a matrix of 5x10 elements where the first element is `m[0][0]` and last one is `m[4][9]`. The first element of the 4th row would be `m[3][0]`.

Strings

A string represents a sequence of characters equivalent to an array of char. It is declared like this:

```
string_name : string[length]
```

The specifier `length` is a number of characters the string consists of. The string is stored internally as the given sequence of characters plus a final `null` character (zero) which is introduced to terminate the string. It does not count against the string's total length.

A null string (`''`) is stored as a single null character.

You can assign string literals or other strings to string variables. String on the right side of an assignment operator has to be shorter or of equal length than the one on the right side. For example:

```
var
  msg1 : string[20];
  msg2 : string[19];

begin
  msg1 := 'This is some message';
  msg2 := 'Yet another message';

  msg1 := msg2; // this is ok, but vice versa would be illegal
```

Alternately, you can handle strings element–by–element. For example:

```
var s : string[5];
...
s := 'mik';
{
s[0] is char literal 'm'
s[1] is char literal 'i'
s[2] is char literal 'k'
s[3] is zero
s[4] is undefined
s[5] is undefined
}
```

Be careful when handling strings in this way, since overwriting the end of a string will cause an unpredictable behavior.

String Concatenating

mikroPascal PRO for PIC32 allows you to concatenate strings by means of plus operator. This kind of concatenation is applicable to string variables/literals, character variables/literals. For control characters, use the non-quoted hash sign and a numeral (e.g. `#13` for CR).

```
var msg : string[20];
    res_txt : string[5];
    res, channel : word;

begin
  //...

  // Get result of ADC
  res := Adc_Read(channel);

  // Create string out of numeric result
  WordToStr(res, res_txt);

  // Prepare message for output
  msg := 'Result is ' +      // Text "Result is"
         res_txt     ;      // Result of ADC

  //...
```

Notes :

- In current version plus operator for concatenating strings will accept at most two operands.
- mikroPascal PRO for PIC32 includes a String Library which automatizes string related tasks.

Pointers

A pointer is a data type which holds a memory address. While a variable accesses that memory address directly, a pointer can be thought of as a reference to that memory address.

To declare a pointer data type, add a carat prefix (`^`) before type. For example, in order to create a pointer to an `integer`, write:

```
^integer;
```

In order to access data at the pointer's memory location, add a carat after the variable name. For example, let's declare variable `p` which points to a `word`, and then assign value 5 to the pointed memory location:

```
var p : ^word;
...
p^ := 5;
```

A pointer can be assigned to another pointer. However, note that only the address, not the value, is copied. Once you modify the data located at one pointer, the other pointer, when dereferenced, also yields modified data.

Pointers and memory spaces

Pointers can point to data in any available memory space.
Pointers can reside in any available memory space except in program (code) memory space.

```
var ptr1: ^const byte;                        // ptr1 pointer in data space pointing to a
byte in code space
var ptr2: ^const ^volatile sfr byte; rx; // ptr2 is pointer in rx space pointing to a
pointer in code space pointing to volatile byte in sfr space
var ptr3: ^data byte; code;                  // error, pointers can not be placed in code
space
```

Due to backward compatibility, pointers to program memory space can also be declared within constant declaration block (using keyword const):

```
program const_ptr;

// constant array will be stored in program memory
const b_array: array[5] of byte = (1,2,3,4,5);

const ptr: ^byte;     // ptr is pointer to program memory space

begin
  ptr   := @b_array;  // ptr now points to b_array[0]
  PORTA := ptr^;
  ptr   := ptr + 3;   // ptr now points to b_array[3]
  PORTA := ptr^;
end.
```

This leads to equality of the following declarations:

```
var   ptr1 : ^const byte;   // ptr1 pointer in data space pointing to a byte in code
space
const ptr2 : ^byte;         // ptr2 pointer in data space pointing to a byte in code
space
```

Therefore, when declaring a pointer within constant declaration block, const qualifier refers to pointed object, not to pointer itself.

Notes :

- Pointer to constant space (Flash memory) is allocated in RAM.
- Constants of a simple type are not allocated in the Flash memory nor in RAM, but changed in the compile time, and therefore address of a such constant can not be obtained.

Function Pointers

Function pointers are allowed in mikroPascal PRO for PIC32. The example shows how to define and use a function pointer:

Example:

Example demonstrates the usage of function pointers. It is shown how to declare a procedural type, a pointer to function and finally how to call a function via pointer.

```
program Example;

type TMyFunctionType = function (param1, param2: byte; param3: word) : word;  // First, 
define the procedural type
var MyPtr: ^TMyFunctionType;          // This is a pointer to previously defined type
    Sample: word;

function Func1(p1, p2: byte; p3: word): word;              // Now, define few functions 
which will be pointed to. Make sure that parameters match the type definition
begin
    result := p1 and p2 or p3;        // return something
end;

function Func2(abc: byte; def: byte; ghi: word): word;     // Another function of the 
same kind. Make sure that parameters match the type definition
begin
  result := abc * def + ghi;          // return something
end;

function Func3(first, yellow: byte; monday: word): word;   // Yet another function. Make 
sure that parameters match the type definition
begin
  result := monday - yellow - first;  // return something
end;

//  main program:
begin
    MyPtr  := @Func1;                  // MyPtr now points to Func1
    Sample := MyPtr^(1, 2, 3);         // Perform function call via pointer, call Func1, 
the return value is 3
    MyPtr  := @Func2;                  // MyPtr now points to Func2
    Sample := MyPtr^(1, 2, 3);         // Perform function call via pointer, call Func2, 
the return value is 5
    MyPtr  := @Func3;                  // MyPtr now points to Func3
    Sample := MyPtr^(1, 2, 3);         // Perform function call via pointer, call Func3, 
the return value is 0
end.
```

Therefore, when declaring a pointer within constant declaration block, const qualifier refers to pointed object, not to pointer itself.

@ Operator

The @ operator constructs a pointer to its operand. The following rules are applied to @:

- If `X` is a variable, `@X` returns a pointer to `X`.

Note : If variable `X` is of array type, the `@` operator will return pointer to it's first basic element, except when the left side of the statement in which `X` is used is an array pointer.
In this case, the `@` operator will return pointer to array, not to it's first basic element.

```
program example;

var w       :  word;
    ptr_b   : ^byte;
    ptr_arr : ^array[10] of byte;
    arr     :  array[10] of byte;

begin
  ptr_b   := @arr; // @ operator will return ^byte
  w       := @arr; // @ operator will return ^byte
  ptr_arr := @arr; // @ operator will return ^array[10] of byte
end.
```

- If F is a routine (a function or procedure), `@F` returns a pointer to `F`.

Related topics: Pointer Arithmetic

Pointer Arithmetic

Pointer arithmetic in the mikroPascal PRO for PIC32 is limited to:

- assigning one pointer to another,
- comparing two pointers,
- comparing pointer to zero,
- adding/subtracting pointer and an integer value,
- subtracting two pointers.

Assignment and Comparison

The simple assignment operator (`=`) can be used to assign value of one pointer to another if they are of the same type.

Assigning the integer constant 0 to a pointer assigns a null pointer value to it.

Two pointers pointing to the same array may be compared by using relational operators `=`, `<>`, `<`, `<=`, `>`, and `>=`. Results of these operations are the same as if they were used on subscript values of array elements in question:

```
var ptr1 : ^byte;
    ptr2 : ^byte;
    a : array[10] of byte;    // array a containing 10 elements of type byte

begin
  ptr1 := @a[4];
  ptr2 := @a[2];

  if (ptr1 = ptr2) then ...   // won't be executed as 4 is not equal to 2
  if (ptr1 > ptr2) then ...   // will be executed as 4 is greater than 2

  if (ptr1^ = ptr2^) then ... // if the value pointed to by ptr1 is equal to the value
pointed to by ptr2 ...
  if (ptr1^ > ptr2^) then ... // if the value pointed to by ptr1 is greater to the value
pointed to by ptr2 ...
end.
```

Note : Comparing pointers pointing to different objects/arrays can be performed at programmer's own responsibility — a precise overview of data's physical storage is required.

Pointer Addition

You can use `Inc` to add an integral value to a pointer. The result of addition is defined only if the pointer points to an element of an array and if the result is a pointer pointing to the same array (or one element beyond it).

If a pointer is declared to point to type, adding an integral value n to the pointer increments the pointer value by `n * sizeof(type)` as long as the pointer remains within the legal range (first element to one beyond the last element). If type has a size of 10 bytes, then adding 5 to a pointer to type advances the pointer 50 bytes in memory.

For example:

```
var
  a : array[10] of byte;    // array a containing 10 elements of type byte
  ptr : ^byte;              // pointer to byte

begin
  ptr := @a[0];        // ptr is pointer to byte, pointing to a[0]
  ptr := ptr + 3;      // ptr+3 is a pointer pointing to a[3]
  ptr^ := 6;      // a[3] now equals 6
  Inc(ptr);            // ptr now points to the next element of array a: a[4]
end.
```

Also, you may sum values pointed to by pointers.

For example:

```
var
  i, j, x :  byte; // variables
  ptr1 : ^byte;    // pointers to byte
  ptr2 : ^byte;
begin
  i := 10;        // assign value 10 to variable; i is at the address 0x0038
```

```
  j := 5;          // assign value 10 to variable; j is at the address 0x003A

  ptr1 := @i;      // ptr1 is pointer to byte, pointing to i
  ptr2 := @j;      // ptr2 is a pointer pointing to j

  x := ptr1^ + ptr2^;   // result is equal to the sum of the values pointed to; x = 5
end.
```

Pointer Subtraction

Similar to addition, you can use Dec to subtract an integral value from a pointer.

If a pointer is declared to point to type, subtracting an integral value n from the the pointer decrements the pointer value by `n * sizeof(type)` as long as the pointer remains within the legal range (first element to one beyond the last element). If type has a size of 10 bytes, then subtracting 5 from a pointer to type pushes back the pointer 50 bytes in memory.

For example:

```
var
  a : array[10] of byte;     // array a containing 10 elements of type byte
  ptr : ^byte;               // pointer to byte

begin
  ptr := @a[6];        // ptr is pointer to byte, pointing to a[6]
  ptr := ptr - 3;      // ptr-3 is a pointer pointing to a[3]
  ptr^ := 6;      // a[3] now equals 6
  Dec(ptr);            // ptr now points to the previous element of array a: a[2]
end.
```

Also, you may subtract two pointers. The difference will be equal to the distance between two pointed addresses, and is calculated regarding to the type which the pointer points to.

For example:

```
var
  i, j, x :  byte; // variables
  ptr1 : ^byte;    // pointers to byte
  ptr2 : ^byte;

begin
  i := 10;         // assign value 10 to variable; i is at the address 0x0039
  j := 5;          // assign value 5 to variable;  j is at the address 0x003A

ptr1 := @i;        // ptr1 is a pointer to byte, pointing to i
  ptr2 := @j;      // ptr2 is a pointer pointing to j

  x := ptr2 - ptr1;     // result is equal to the distance between the two pointed ad-
dresses; x = 1 (1 byte)
  x := ptr1^ - ptr2^;   // result is equal to the difference of the values pointed to;
x = 5
end.
```

Records

A record (analogous to a structure in some languages) represents a heterogeneous set of elements. Each element is called a field. The declaration of the record type specifies a name and type for each field. The syntax of a record type declaration is

```
type recordTypeName = record
  fieldList1 : type1;
  ...
  fieldListn : typen;
end;
```

where `recordTypeName` is a valid identifier, each type denotes a type, and each `fieldList` is a valid identifier or a comma-delimited list of identifiers. The scope of a field identifier is limited to the record in which it occurs, so you don't have to worry about naming conflicts between field identifiers and other variables.

Note : In mikroPascal PRO for PIC32, you cannot use the record construction directly in variable declarations, i.e. without `type`.

For example, the following declaration creates a record type called `TDot`:

```
type
  TDot = record
    x, y : real;
end;
```

Each `TDot` contains two fields: `x` and `y` coordinates. Memory is allocated when you declare the record, like this:

```
var m, n: TDot;
```

This variable declaration creates two instances of `TDot`, called `m` and `n`.

A field can be of previously defined record type. For example:

```
// Structure defining a circle:
type
  TCircle = record
    radius : real;
    center : TDot;
end;
```

Accessing Fields

You can access the fields of a record by means of dot (`.`) as a direct field selector. If we have declared variables `circle1` and `circle2` of previously defined type TCircle:

```
var circle1, circle2 : TCircle;
```

we could access their individual fields like this:

```
circle1.radius := 3.7;
circle1.center.x := 0;
circle1.center.y := 0;
```

Accessing the fields is possible via the with statement as well.

You can also commit assignments between complex variables, if they are of the same type:

```
circle2 := circle1; // This will copy values of all fields
```

Types Conversions

Conversion of variable of one type to a variable of another type is typecasting. mikroPascal PRO for PIC32 supports both implicit and explicit conversions for built-in types.

Implicit Conversion

Compiler will provide an automatic implicit conversion in the following situations:

- statement requires an expression of particular type (according to language definition), and we use an expression of different type,
- operator requires an operand of particular type, and we use an operand of different type,
- function requires a formal parameter of particular type, and we pass it an object of different type,
- `result` does not match the declared function return type.

Promotion

When operands are of different types, implicit conversion promotes the less complex type to more complex type taking the following steps:

```
bit       →  byte/char
byte/char →  word
short     →  integer
short     →  longint
integer   →  longint
integer   →  real
```

Higher bytes of extended unsigned operand are filled with zeroes. Higher bytes of extended signed operand are filled with bit sign (if number is negative, fill higher bytes with one, otherwise with zeroes). For example:

```
var a : byte; b : word;
...
a := $FF;
b := a;  // a is promoted to word, b becomes $00FF
```

Clipping

In assignments and statements that require an expression of particular type, destination will store the correct value only if it can properly represent the result of expression, i.e. if the result fits in destination range.

If expression evaluates to a more complex type than expected, excess of data will be simply clipped (higher bytes are lost).

```
var i : byte; j : word;
//...
j := $FF0F;
i := j;   // i becomes $0F, higher byte $FF is lost
```

Explicit Conversion

Explicit conversion can be executed at any point by inserting type keyword (byte, word, short, integer, longint or real) ahead of an expression to be converted. The expression must be enclosed in parentheses. Explicit conversion can be performed only on the operand right of the assignment operator.

Special case is conversion between signed and unsigned types. Explicit conversion between signed and unsigned data does not change binary representation of data — it merely allows copying of source to destination.

For example:

```
var a : byte; b : short;
...
b := -1;
a := byte(b);  // a is 255, not 1

// This is because binary representation remains
// 11111111; it's just interpreted differently now
```

You can't execute explicit conversion on the operand left of the assignment operator:

```
word(b) := a;   // Compiler will report an error
```

Conversions Examples

Here is an example of conversion:

```
program test;

type TBytePtr = ^byte;

var arr: array[10] of word;
    ptr : TBytePtr;

var a, b, cc : byte;
          dd : word;
```

```
begin
  a := 241;
  b := 128;

  cc  := a + b;         // equals 113
  cc  := word(a + b);  // equals 113
  dd  := a + b;         // equals 369

  ptr := TBytePtr(@arr);
  ptr := ^byte(@arr);
end.
```

Type Specifier

The specifier type introduces a synonym for a specified type. The type declarations are used to construct shorter or more convenient names for types already defined by the language or declared by the user.

The specifier type stands first in the declaration:

```
type synonym = <type_definition>;
```

The type keyword assigns `synonym` to <type_definition>. The synonym needs to be a valid identifier.

A declaration starting with the type specifier does not introduce an object or a function of a given type, but rather a new name for a given type. In other words, the `type` declaration is identical to a “normal” declaration, but instead of objects, it declares types. It is a common practice to name custom type identifiers with starting capital letter — this is not required by the mikroPascal PRO for PIC32.

For example:

```
// Let’s declare a synonym for “byte”
type Distance = byte;

// Now, synonym “Distance” can be used as type identifier:
var i : Distance; // declare variable i of byte
```

Type Qualifiers

The type qualifiers const and volatile are optional in declarations and do not actually affect the type of declared object.

Qualifier const

The qualifier `const` implies that a declared object will not change its value during runtime. In declarations with the `const` qualifier all objects need to be initialized.

The mikroPascal PRO for PIC32 treats objects declared with the `const` qualifier the same as literals or preprocessor constants. If the user tries to change an object declared with the const qualifier compiler will report an error.

For example:

```
const PI : byte := 3.14159;
```

Qualifier volatile

The qualifier volatile implies that a variable may change its value during runtime independently from the program. Use the volatile modifier to indicate that a variable can be changed by a background routine, an interrupt routine, or I/O port. Declaring an object to be volatile warns the compiler not to make assumptions concerning the value of an object while evaluating expressions in which it occurs because the value could be changed at any moment.

Operators

Operators are tokens that trigger some computation when being applied to variables and other objects in an expression.

There are four types of operators in mikroPascal PRO for PIC32:

- Arithmetic Operators
- Bitwise Operators
- Boolean Operators
- Relational Operators

Operators Precedence and Associativity

There are 4 precedence categories in mikroPascal PRO for PIC32. Operators in the same category have equal precedence with each other.

Each category has an associativity rule: left-to-right (→), or right-to-left (←). In the absence of parentheses, these rules resolve the grouping of expressions with operators of equal precedence.

Precedence	Operands	Operators	Associativity
4	1	`@ not + -`	←
3	2	`* / div mod and shl shr`	→
2	2	`+ - or xor`	→
1	2`	`= <> < > <= >=`	→

Arithmetic Operators

Arithmetic operators are used to perform mathematical computations. They have numerical operands and return numerical results. Since the char operators are technically bytes, they can be also used as unsigned operands in arithmetic operations.

All arithmetic operators associate from left to right.

Operator	Operation	Operands	Result
`+`	addition	`byte, short, word, integer, longint, dword, real`	`byte, short, word, integer, longint, dword, real`
`-`	subtraction	`byte, short, word, integer, longint, dword, real`	`byte, short, word, integer, longint, dword, real`
`*`	multiplication	`byte, short, word, integer, longint, dword, real`	`word, integer, longint, dword, real`
`/`	division, floating-point	`byte, short, word, integer, longint, dword, real`	`real`
`div`	division, rounds down to nearest integer	`byte, short, word, integer, longint, dword`	`byte, short, word, integer, longint, dword`
`mod`	modulus, returns the remainder of integer division (cannot be used with floating points)	`byte, short, word, integer, longint, dword`	`byte, short, word, integer, longint, dword`

Division by Zero

If 0 (zero) is used explicitly as the second operand (i.e. `x div 0`), the compiler will report an error and will not generate code.
But in case of implicit division by zero: `x div y`, where y is 0 (zero), the result will be the maximum integer (i.e `255`, if the result is byte type; `65536`, if the result is word type, etc.).

Unary Arithmetic Operators

Operator - can be used as a prefix unary operator to change sign of a signed value. Unary prefix operator + can be used, but it doesn't affect data.

For example:

```
b := -a;
```

Relational Operators

Use relational operators to test equality or inequality of expressions. All relational operators return `TRUE` or `FALSE`.

All relational operators associate from left to right.

Relational Operators Overview

Operator	Operation
=	equal
<>	not equal
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal

Relational Operators in Expressions

Precedence of arithmetic and relational operators is designated in such a way to allow complex expressions without parentheses to have expected meaning:

```
a + 5 >= c - 1.0 / e   // → (a + 5) >= (c - (1.0 / e))
```

Bitwise Operators

Use bitwise operators to modify individual bits of numerical operands.

Bitwise operators associate from left to right. The only exception is the bitwise complement operator not which associates from right to left.

Bitwise Operators Overview

Operator	Operation
`and`	bitwise AND; compares pairs of bits and returns 1 if both bits are 1, otherwise it returns 0
`or`	bitwise (inclusive) OR; compares pairs of bits and generates a 1 result if either or both bits are 1, otherwise it returns 0
`xor`	bitwise exclusive OR (XOR); compares pairs of bits and generates a 1 result if the bits are complementary, otherwise it returns 0
`not`	bitwise complement (unary); inverts each bit
`shl`	bitwise shift left; moves the bits to the left, discards the far left bit and assigns 0 to the right most bit.
`shr`	bitwise shift right; moves the bits to the right, discards the far right bit and if unsigned assigns 0 to the left most bit, otherwise sign extends

Logical Operations on Bit Level

and	**0**	**1**
0	0	0
1	0	1

or	**0**	**1**
0	0	1
1	1	1

xor	**0**	**1**
0	0	1
1	1	0

not	**0**	**1**
	1	0

Bitwise operators `and`, `or`, and `xor` perform logical operations on the appropriate pairs of bits of their operands. The operator not complements each bit of its operand. For example:

```
$1234 and $5678         // equals $1230

{ because ..

$1234 : 0001 0010 0011 0100
$5678 : 0101 0110 0111 1000
----------------------------
 and  : 0001 0010 0011 0000

.. that is, $1230 }


// Similarly:

$1234 or  $5678         // equals $567C
$1234 xor $5678         // equals $444C
not $1234               // equals $EDCB
```

Unsigned and Conversions

If a number is converted from less complex to more complex data type, the upper bytes are filled with zeroes. If a number is converted from more complex to less complex data type, the data is simply truncated (the upper bytes are lost).

For example:

```
var a : byte; b : word;
...
  a := $AA;
  b := $F0F0;
  b := b and a;
  { a is extended with zeroes; b becomes $00A0 }
```

Signed and Conversions

If number is converted from less complex to more complex data type, the upper bytes are filled with ones if sign bit is 1 (number is negative); the upper bytes are filled with zeroes if sign bit is 0 (number is positive). If number is converted from more complex to less complex data type, the data is simply truncated (the upper bytes are lost).

For example:

```
var a : byte; b : word;
...
  a := -12;
  b := $70FF;
  b := b and a;

  { a is sign extended, with the upper byte equal to $FF;
    b becomes $70F4 }
```

Bitwise Shift Operators

Binary operators `shl` and `shr` move the bits of the left operand by a number of positions specified by the right operand, to the left or right, respectively. Right operand has to be positive and less than 255.

With shift left (`shl`), left most bits are discarded, and “new” bits on the right are assigned zeroes. Thus, shifting unsigned operand to the left by n positions is equivalent to multiplying it by 2n if all discarded bits are zero. This is also true for signed operands if all discarded bits are equal to the sign bit.

With shift right (`shr`), right most bits are discarded, and the “freed” bits on the left are assigned zeroes (in case of unsigned operand) or the value of the sign bit (in case of signed operand). Shifting operand to the right by n positions is equivalent to dividing it by 2^n.

Boolean Operators

Although mikroPascal PRO for PIC32 does not support boolean type, you have Boolean operators at your disposal for building complex conditional expressions. These operators conform to standard Boolean logic and return either `TRUE` (all ones) or `FALSE` (zero):

Operator	Operation
`and`	logical AND
`or`	logical OR
`xor`	logical exclusive OR (XOR)
`not`	logical negation

Boolean operators associate from left to right. Negation operator not associates from right to left.

Unary Operators

Unary operators are operators that take exactly one argument.

Unary Arithmetic Operator

Operator - can be used as a prefix unary operator to change sign of a signed value. Unary prefix operator + can be used also, but it doesn't affect data.

For example:

```
b := -a;
```

Unary Bitwise Operator

The result of the not (bitwise negation) operator is the bitwise complement of the operand. In the binary representation of the result, every bit has the opposite value of the same bit in the binary representation of the operand.

Operator	Operation
`not`	bitwise complement (unary); inverts each bit

Example:

```
not 0x1234                ' equals 0xEDCB
```

Address and Indirection Operator

In the mikroPascal PRO for PIC32, address of an object in memory can be obtained by means of an unary operator `@`. To reach the pointed object, we use an indirection operator `^` on a pointer. See Pointers section for more details.

Operator	Operation
^	accesses a value indirectly, through a pointer; result is the value at the address to which operand points
@	constructs a pointer to its operand

See Pointers for more details on this subject

Note : Besides these, sizeof and explicit conversion unary operators are supported also.

Sizeof Operator

The prefix unary operator sizeof returns an integer constant that represents the size of memory space (in bytes) used by its operand (determined by its type, with some exceptions).

The operator sizeof can take either a type identifier or an unary expression as an operand. You cannot use sizeof with expressions of function type, incomplete types, parenthesized names of such types, or with lvalue that designates a bit field object.

Sizeof Applied to Expression

If applied to expression, the size of an operand is determined without evaluating the expression (and therefore without side effects). The result of the operation will be the size of the type of the expression's result.

Sizeof Applied to Type

If applied to a type identifier, sizeof returns the size of the specified type. The unit for type size is sizeof(byte) which is equivalent to one byte.

Thus:

```
sizeof(byte)            // returns 1
sizeof(integer)         // returns 2
sizeof(dword)           // returns 4
sizeof(real)            // returns 4
```

When the operand is a non-parameter of array type, the result is the total number of bytes in the array (in other words, an array name is not converted to a pointer type):

```
var  i, j : integer;
     samples : array[10] of integer;
...
j := sizeof(samples[1]);  // j = sizeof(integer) = 2
i := sizeof(samples);     // i = 10*sizeof(integer) = 20
```

If the operand is a parameter declared as array type or function type, `sizeof` gives the size of the pointer. When applied to records, `sizeof` gives the total number of bytes, including any padding. The operator `sizeof` cannot be applied to a function.

Expressions

An expression is a sequence of operators, operands and punctuators that returns a value.

The primary expressions include: literals, constants, variables and function calls. More complex expressions can be created from primary expressions by using operators. Formally, expressions are defined recursively: subexpressions can be nested up to the limits of memory.

Expressions are evaluated according to certain conversion, grouping, associativity and precedence rules which depend on the operators in use, presence of parentheses and data types of the operands. The precedence and associativity of the operators are summarized in Operator Precedence and Associativity. The way operands and subexpressions are grouped does not necessarily specify the actual order in which they are evaluated by mikroPascal PRO for PIC.

Expression Evaluation

General Rule

Expression are evaluated according to the right side operands. Operations are done at higher operand level, with signed operands taking precedence.

Example :

```
a : byte;
b : word;
c : integer;

a * b // word level
a * c // integer level
b * c // integer level
```

Left side exception

In arithmetic expression left side is considered in the following manner : If the left side size in bytes is greater than higher operand size, then evaluation is done at one level above higher operand level (to get correct calculations).

Example :

```
a: dword;
b: byte;

a := b * 5; // this is done at word level
```

Conditional expressions

Conditional expressions may differ from the same code in assignment expressions (due to left side exception).

Example :

```
a: dword;
b: byte

if b*5 then... // byte level - general rule will not give same result as

a := b * 5      // word level - general rule + left side exception
if a then...

if b*5 exceeds byte range.
```

Explicit Typecasting

Any expression can be evaluated at specific level by using explicit typecasting. Having in mind previous example, in order to get same calculation in conditional and assignment expression, the following should be done :

```
if word(b*5) then... // word level
```

Statements

Statements define algorithmic actions within a program. Each statement needs to be terminated with a semicolon (;). In the absence of specific jump and selection statements, statements are executed sequentially in the order of appearance in the source code.

The most simple statements are assignments, procedure calls and jump statements. These can be combined to form loops, branches and other structured statements.

Refer to:

- Assignment Statements
- Compound Statements (Blocks)
- Conditional Statements
- Iteration Statements (Loops)
- Jump Statements

- asm Statement

Assignment Statements

Assignment statements have the following form:

```
variable := expression;
```

The statement evaluates `expression` and assigns its value to variable. All the rules of implicit conversion are applied. `Variable` can be any declared variable or array element, and expression can be any expression.

Do not confuse the assignment with relational operator = which tests for equality. Also note that, although similar, the construction is not related to the declaration of constants.

Compound Statements (Blocks)

Compound statement, or block, is a list of statements enclosed by keywords `begin` and end:

```
begin
  statements
end;
```

Syntactically, a block is considered to be a single statement which is allowed to be used when Pascal syntax requires a single statement. Blocks can be nested up to the limits of memory.

For example, the `while` loop expects one statement in its body, so we can pass it a compound statement:

```
while i < n do
  begin
    temp := a[i];
    a[i] := b[i];
    b[i] := temp;
    i := i + 1;
  end;
```

Conditional Statements

Conditional or selection statements select one of alternative courses of action by testing certain values. There are two types of selection statements:

- if
- case

If Statement

Use the keyword if to implement a conditional statement. The syntax of the if statement has the following form:

```
if expression then statement1 [else statement2]
```

If `expression` evaluates to true then `statement1` executes. If expression is false then `statement2` executes. The expression must convert to a boolean type; otherwise, the condition is ill-formed. The else keyword with an alternate statement (`statement2`) is optional.

There should never be a semicolon before the keyword `else`.

Nested if statements

Nested if statements require additional attention. A general rule is that the nested conditionals are parsed starting from the innermost conditional, with each `else` bound to the nearest available `if` on its left:

```
if expression1 then
if expression2 then statement1
else statement2
```

The compiler treats the construction in this way:

```
if expression1 then
begin
  if expression2 then statement1
  else statement2
end
```

In order to force the compiler to interpret our example the other way around, we have to write it explicitly:

```
if expression1 then
begin
  if expression2 then statement1
end
else statement2
```

Case Statement

Use the `case` statement to pass control to a specific program branch, based on a certain condition. The case statement consists of a selector expression (a condition) and a list of possible values. The syntax of the case statement is:

```
case selector of
  value_1 : statement_1
  ...
  value_n : statement_n
  [else default_statement]
end;
```

`selector` is an expression which should evaluate as integral value. values can be literals, constants, or expressions, and `statements` can be any statements.

The `else` clause is optional. If using the else branch, note that there should never be a semicolon before the keyword else.

First, the `selector` expression (condition) is evaluated. Afterwards the case statement compares it against all available `values`. If the match is found, the statement following the match evaluates, and the `case` statement terminates. In case there are multiple matches, the first matching statement will be executed. If none of values matches selector, then `default_statement` in the else clause (if there is some) is executed.

Here's a simple example of the case statement:

```
case operator of
  '*' : result := n1 * n2;
  '/' : result := n1 / n2;
  '+' : result := n1 + n2;
  '-' : result := n1 - n2
else result := 0;
end;
```

Also, you can group values together for a match. Simply separate the items by commas:

```
case reg of
  0:       opmode := 0;
  1,2,3,4: opmode := 1;
  5,6,7:   opmode := 2;
end;
```

In mikroPascal PRO for PIC32, values in the case statement can be variables too:

```
case byte_variable of

  byte_var1: opmode := 0;  // this will be compiled correctly

  byte_var2:
             opmode := 1;  // avoid this case, compiler will parse
                           // a variable followed by colon sign as label

  byte_var3: //               adding a comment solves the parsing problem
             opmode := 2;
end;
```

Nested Case statement

Note that the case statements can be nested – values are then assigned to the innermost enclosing case statement.

Iteration Statements

Iteration statements let you loop a set of statements. There are three forms of iteration statements in mikroPascal PRO for PIC32:

- for
- while...do
- repeat

You can use the statements break and continue to control the flow of a loop statement. break terminates the statement in which it occurs, while continue begins executing the next iteration of the sequence.

For Statement

The for statement implements an iterative loop and requires you to specify the number of iterations. The syntax of the for statement is:

```
for counter := initial_value to final_value do statement_list
// or
for counter := initial_value downto final_value do statement_list
```

`counter` is a variable which increments (or decrements if you use `downto`) with each iteration of the loop. Before the first iteration, `counter` is set to `initial_value` and will increment (or decrement) until it reaches `final_value`. `final_value` will be recalculated each time the loop is reentered.
This way number of loop iterations can be changed inside the loop by changing final_value. With each iteration, `statement_list` will be executed.

`initial_value` and `final_value` should be expressions compatible with counter.

If `final_value` is a complex expression whose value can not be calculated in compile time and number of loop iterations is not to be changed inside the loop by the means of `final_value`, it should be calculated outside the for statement and result should be passed as for statement's `final_value`. `statement_list` is a list of statements that do not change the value of counter. If `statement_list` contains more than one statement, statements must be enclosed within **`begin-end`** block.

Here is an example of calculating scalar product of two vectors, a and b, of length 10, using the for statement:

```
s := 0;
for i := 0 to 9 do
  s := s + a[i] * b[i];
```

Endless Loop

The for statement results in an endless loop if `final_value` equals or exceeds the range of the `counter`'s type.

More legible way to create an endless loop in Pascal is to use the statement **`while`** `TRUE` **`do`**.

While Statement

Use the while keyword to conditionally iterate a statement. The syntax of the while statement is:

```
while expression do statement
```

`statement` is executed repeatedly as long as `expression` evaluates true. The test takes place before the `statement` is executed. Thus, if `expression` evaluates false on the first pass, the loop does not execute.

Here is an example of calculating scalar product of two vectors, using the `while` statement:

```
s := 0; i := 0;
while i < n do
begin
  s := s + a[i] * b[i];
  i := i + 1;
end;
```

Probably the easiest way to create an endless loop is to use the statement:

```
while TRUE do ...;
```

Repeat Statement

The repeat statement executes until the condition becomes true. The syntax of the repeat statement is:

```
repeat statement until expression
```

`statement` is executed repeatedly as long as `expression` evaluates false. The expression is evaluated after each iteration, so the loop will execute `statement` at least once.

Here is an example of calculating scalar product of two vectors, using the `repeat` statement:

```
s := 0; i := 0;
...
repeat
  begin
    s := s + a[i] * b[i];
    i := i + 1;
  end;
until i = n;
```

Jump Statements

The jump statement, when executed, transfers control unconditionally. There are four such statements in mikroPascal PRO for PIC32:

- break
- continue
- exit
- goto

asm Statement

mikroPascal PRO for PIC32 allows embedding assembly in the source code by means of the asm statement. Note that you cannot use numerals as absolute addresses for register variables in assembly instructions. You may use symbolic names instead (listing will display these names as well as addresses).

You can group assembly instructions with the `asm` keyword:

```
asm
   block of assembly instructions
end;
```

The only types whose name remains the same in asm as it is in the mikroPascal PRO for PIC32 are registers, e.g. INTCON, PORTB, WREG, GIE, etc.

mikroPascal PRO for PIC32 comments are allowed in embedded assembly code.

Accessing variables

Depending on the place of declaration, accessing a variable can be done in several ways :

- **Accessing global variable** :
1. If declared under implementation section (visible only in the file where it was declared) : `<source_file_name>_<variable_name>`.
2. If declared in the interface section (visible throughout the whole project) : `_<variable_name>`.
3. If accessing registers (declared through register, rx or sfr specifiers, visible throughout the whole project) : `<variable_name>`.
- **Accessing local variable** : `<routine_name>_<variable_name>`.
- **Accessing routine parameter** : `FARG_<routine_name>_<variable_name>`.

Here is an example of using asm instructions :

```
program asm_example;

var myvar : word; absolute 0x2678;
const msg = 'Hello'; org 0x3678;
var myvar1 : dword;

procedure proc(); org 0x1234;
begin
  asm
    nop
  end;
end;

begin
  myvar := 5;
  myvar1 := 0xABCD1234;

  asm
      MOV _myvar, w0                     ; move myvar to W0
      nop
      MOV #6, W0                         ; move literal 6 to W0
      MOV W0, _myvar                     ; move contents of W0 to myvar
      MOV #lo_addr(_myvar), w1          ; retrieve low address word of _myvar and move
it to W1 (0x2678 -> W1)
      MOV #hi_addr(_myvar), W1            ; retrieve high address word of _myvar and
move it to W1 (0x0000 -> W1)
      MOV #lo_addr(_proc), W0           ; retrieve hi address byte of routine proc and
move it to W0 (0x0001 -> W1)
      MOV #lo_addr(_msg), W0              ; retrieve low address word of constant msg
and move it to W0 (0x3652 -> W1)
      MOV _myvar1+2, w0                   ; accessing hi word of myvar1 variable and
move it to W1 (0xABCD -> W1)
  end;
end.
```

Asm code and SSA optimization

If asm code is mixed with the Pascal code, keep in mind that the generated code can substantially differ when SSA optimization option is enabled or disabled.
This is due to the fact that SSA optimization uses certain working registers to store routine parameters (W10-W13), rather than storing them onto the function frame.

Because of this, user must be very careful when writing asm code as existing values in the working registers used by SSA optimization can be overwritten.
To avoid this, it is recommended that user includes desired asm code in a separate routine.

With Statement

The With statement is a convenient method for referencing elements of a complex variable, such as a record.
It simplifies the code by removing the need to prefix each referenced element with the complex variable name; i.e. accessing all of the record's fields with only one reference.

Example:

```
program With_Test;

type Circle_Parameters =
  Record
    x_center : integer;
    y_center : integer;
    radius   : integer;
  end;

var Circle : Circle_Parameters;

begin
  With Circle do
  begin
    x_center := 50;
    y_center := 60;
    radius   := 10;
  end;
end.
```

Directives

Directives are words of special significance which provide additional functionality regarding compilation and output.

The following directives are at your disposal:

- Compiler directives for conditional compilation,
- Linker directives for object distribution in memory.

Compiler Directives

mikroPascal PRO for PIC32 treats comments beginning with a "`$`" immediately following an opening brace as a compiler directive; for example, {`$ELSE`}. The compiler directives are not case sensitive.

You can use a conditional compilation to select particular sections of code to compile, while excluding other sections. All compiler directives must be completed in the source file in which they have begun.

Directives $DEFINE and $UNDEFINE

Use directive `$DEFINE` to define a conditional compiler constant (“flag”). You can use any identifier for a flag, with no limitations. No conflicts with program identifiers are possible because the flags have a separate name space. Only one flag can be set per directive.

For example:

```
{$DEFINE Extended_format}
```

Use `$UNDEFINE` to undefine (“clear”) previously defined flag.

Note : Pascal does not support macros; directives `$DEFINE` and `$UNDEFINE` do not create/destroy macros. They only provide flags for directive `$IFDEF` to check against.

Directives $IFDEF, $IFNDEF, $ELSE and $ENDIF

Conditional compilation is carried out by the `$IFDEF` and `$IFNDEF` directives. `$IFDEF` tests whether a flag is currently defined, and `$IFNDEF` if the flag is not defined, i.e. whether a previous `$DEFINE` directive has been processed for that flag and is still in force.

Directives `$IFDEF` and `$IFNDEF` are terminated with the `$ENDIF` directive and can have an optional `$ELSE` clause:

```
{$IFDEF flag}
  <block of code>
{$ELSE}
  <alternate block of code>
{$ENDIF}
```

First, `$IFDEF` checks if flag is defined by means of `$DEFINE`. If so, only `<block of code>` will be compiled. Otherwise, `<alternate block of code>` will be compiled. `$ENDIF` ends the conditional sequence. The result of the preceding scenario is that only one section of code (possibly empty) is passed on for further processing.

The processed section can contain further conditional clauses, nested to any depth; each `$IFDEF` must be matched with a closing `$ENDIF`.

Here is an example:

```
// Uncomment the appropriate flag for your application:
//{$DEFINE resolution10}
//{$DEFINE resolution12}

{$IFDEF resolution10}
  // <code specific to 10-bit resolution>
{$ELSE}
  {$IFDEF resolution12}
    // <code specific to 12-bit resolution>
  {$ELSE}
    // <default code>
  {$ENDIF}
{$ENDIF}
```

Unlike `$IFDEF`, `$IFNDEF` checks if flag is not defined by means of `$DEFINE`, thus producing the opposite results.

Include Directive $I

The $I parameter directive instructs mikroPascal PRO for PIC32 to include the named text file in the compilation. In effect, the file is inserted in the compiled text right after the `{$I filename}` directive. If filename does not specify a directory path, then, in addition to searching for the file in the same directory as the current unit, mikroPascal PRO for PIC32 will search for file in order specified by the search paths.

To specify a filename that includes a space, surround the file name with quotation marks: `{$I "My file"}`.

There is one restriction to the use of include files: An include file can't be specified in the middle of a statement part. In fact, all statements between the begin and end of a statement part must exist in the same source file.

See also Predefined Project Level Defines.

Linker Directives

mikroPascal PRO for PIC32 uses an internal algorithm to distribute objects within memory. If you need to have a variable, constant or a routine at the specific predefined address, use the linker directives `absolute` and `org`.

When using these directives, be sure to use them in proper memory segments, i.e. for functions it is the KSEG0 and for variables it is the KSEG1. Linker directives are used with the **virtual addresses**.

Directive absolute

Directive absolute specifies the starting address in RAM for a variable. If the variable is multi-byte, higher bytes will be stored at the consecutive locations.

Directive `absolute` is appended to declaration of a variable:

```
// Variable x will occupy 1 word (16 bits) at address 0xA0000000
var x : word; absolute 0xA0000000;

// Variable y will occupy 2 words at addresses 0xA0000000 and 0xA0000002
var y : longint; absolute 0xA0000000;
```

Be careful when using the absolute directive, as you may overlap two variables by accident. For example:

```
// Variable i will occupy 1 word at address 0xA0000002;
var  i : word; absolute 0xA0000002;

// Variable will occupy 2 words at 0xA0000000 and 0xA0000002; thus,
// changing i changes jj at the same time and vice versa
var jj : longint; absolute 0xA0000000;
```

Directive org

Directive org specifies the starting address of a constant or a routine in ROM. It is appended to the constant or a routine declaration.

To place a constant array in Flash memory, write the following :

```
// Constant array MONTHS will be placed starting from the address 0x9D000000
const MONTHS : array[1..12] of byte = (31,28,31,30,31,30,31,31,30,31,30,31); org 0x800;
```

If you want to place simple type constant into Flash memory, instead of following declaration:

```
const SimpleConstant : byte = 0xAA; org 0x9D000000;
```

use an array consisting of single element :

```
const SimpleConstant : array[1] of byte = (0xAA); org 0x9D000000;
```

In first case, compiler will recognize your attempt, but in order to save Flash space, and boost performance, it will automatically replace all instances of this constant in code with it's literal value.
In the second case your constant will be placed in Flash in the exact location specified.

To place a routine on a specific address in Flash memory you should write the following :

```
procedure proc(par : byte); org 0x9D000000;
begin
// Procedure will start at address 0x9D000000;
...
end;
```

`org` directive can be used with `main` routine too. For example:

```
program Led_Blinking;

begin org 0x9D000000;             // main procedure starts at 0x9D000000
...
end.
```

Directive orgall

Use the `orgall` directive to specify the address above which all routines and constants will be placed. Example:

```
begin
  orgall(0x9D000000); // All the routines, constants in main program will be above the address 0x9D000000
  ...
end.
```

CHAPTER 9

mikroPascal PRO for PIC32 Libraries

mikroPascal PRO for PIC32 provides a set of libraries which simplify the initialization and use of PIC32 and their modules:

Use Library manager to include mikroPascal PRO for PIC32 Libraries in you project.

Hardware Libraries

- ADC Library
- CANSPI Library
- Compact Flash Library
- Epson S1D13700 Graphic Lcd Library
- Flash Memory Library
- Graphic Lcd Library
- I²C Library
- Keypad Library
- Lcd Library
- Manchester Code Library
- Memory Manager Library
- Multi Media Card Library
- OneWire Library
- Port Expander Library
- PS/2 Library
- PWM Library
- RS-485 Library
- Software I²C Library
- Software SPI Library
- Software UART Library
- Sound Library
- SPI Library
- SPI Ethernet Library
- SPI Ethernet ENC24J600 Library
- SPI Graphic Lcd Library
- SPI Lcd Library
- SPI Lcd8 Library
- SPI T6963C Graphic Lcd Library
- T6963C Graphic Lcd Library
- TFT Display Library
- Touch Panel Library
- Touch Panel TFT Library
- UART Library
- USB Library

Miscellaneous Libraries

- Button Library
- Conversions Library
- PrintOut Library
- Setjmp Library
- Sprint Library
- Time Library
- Trigonometry Library

See also Built-in Routines.

Hardware Libraries

- ADC Library
- CANSPI Library
- Compact Flash Library
- Epson S1D13700 Graphic Lcd Library
- Flash Memory Library
- Graphic Lcd Library
- I²C Library
- Keypad Library
- Lcd Library
- Manchester Code Library
- Memory Manager Library
- Multi Media Card Library
- OneWire Library
- Port Expander Library
- PS/2 Library
- PWM Library
- RS-485 Library
- Software I²C Library
- Software SPI Library
- Software UART Library
- Sound Library
- SPI Library
- SPI Ethernet Library
- SPI Ethernet ENC24J600 Library
- SPI Graphic Lcd Library
- SPI Lcd Library
- SPI Lcd8 Library
- SPI T6963C Graphic Lcd Library
- T6963C Graphic Lcd Library
- TFT Display Library
- Touch Panel Library
- Touch Panel TFT Library
- UART Library
- USB Library

ADC Library

ADC (Analog to Digital Converter) module is available with a number of PIC32 MCU modules. ADC is an electronic circuit that converts continuous signals to discrete digital numbers. ADC Library provides you a comfortable work with the module.

Library Routines

- ADC1_Init
- ADC1_Init_Advanced
- ADC1_Get_Sample
- ADC1_Read

ADC1_Init

Prototype	`procedure ADC1_Init();`
Description	This routines configures ADC module to work with default settings. The internal ADC module is set to: - single channel conversion - 10-bit conversion resolution - unsigned integer data format - auto-convert - VRef+ : AVdd, VRef- : AVss - instruction cycle clock - conversion clock : 32*Tcy - auto-sample time : 31TAD
Parameters	None.
Returns	Nothing.
Requires	- MCU with built-in ADC module.
Example	`ADC1_Init();  // Initialize ADC module with default settings`
Notes	None.

ADC1_Init_Advanced

Prototype	`procedure ADC1_Init_Advanced(Reference : word);`
Description	This routine configures the internal ADC module to work with user defined settings.

Parameters	- `Reference`: voltage reference used in ADC process.
	
	Description / **Predefined library const**
	Voltage reference:
	`Internal voltage reference` — `_ADC_INTERNAL_REF`
	`External voltage reference` — `_ADC_EXTERNAL_REF`

Returns	Nothing.
Requires	- The MCU with built-in ADC module.
Example	`ADC1_Init_Advanced(_ADC_INTERNAL_REF);  // set internal reference used`
Notes	- Not all MCUs support advanced configuration. Please, read the appropriate datasheet before utilizing this library.

Description	Predefined library const
Voltage reference:	
`Internal voltage reference`	`_ADC_INTERNAL_REF`
`External voltage reference`	`_ADC_EXTERNAL_REF`

ADC1_Get_Sample

Prototype	`function ADC1_Get_Sample(channel : word) : word;`
Description	The function enables ADC module and reads the specified analog channel input.
Parameters	- `channel` represents the channel from which the analog value is to be acquired.
Returns	10-bit unsigned value from the specified `channel`.
Requires	- The MCU with built-in ADC module. - Prior to using this routine, ADC module needs to be initialized. See ADCx_Init and ADCx_Init_Advanced. - Before using the function, be sure to configure the appropriate TRISx bits to designate pins as inputs.

Example	`var adc_value : word;`
	`...`
	`adc_value = ADC1_Get_Sample(10);     // read analog value from ADC module channel 10`

Notes	- The function sets the appropriate bit in the AD1PCFG registers to enable analog function of the chosen pin. - Refer to the appropriate Datasheet for channel-to-pin mapping.

ADC1_Read

Prototype	`function ADC1_Read(channel : word) : word;`
Description	The function initializes, enables ADC module and reads the specified analog channel input.
Parameters	- `channel` represents the channel from which the analog value is to be acquired.
Returns	10-bit unsigned value from the specified `channel`.
Requires	- The MCU with built-in ADC module. - Before using the function, be sure to configure the appropriate TRISx bits to designate pins as inputs.

Example	`var adc_value : word;`
	`...`
	`adc_value = ADC1_Read(10); // read analog value from ADC module channel 10`

Notes	- This is a standalone routine, so there is no need for a previous initialization of ADC module. - The function sets the appropriate bit in the ADPCFG registers to enable analog function of the chosen pin. - Refer to the appropriate Datasheet for channel-to-pin mapping.

Library Example

This code snippet reads analog value from the channel 1 and sends readings as a text over UART1.

Copy Code To Clipboard

```
program Temperature_Sensor;

// LCD module connections
var LCD_RS : sbit  at LATB2_bit;
var LCD_EN : sbit  at LATB3_bit;
var LCD_D4 : sbit  at LATB4_bit;
var LCD_D5 : sbit  at LATB5_bit;
var LCD_D6 : sbit  at LATB6_bit;
var LCD_D7 : sbit  at LATB7_bit;

var LCD_RS_Direction : sbit at TRISB2_bit;
var LCD_EN_Direction : sbit at TRISB3_bit;
var LCD_D4_Direction : sbit at TRISB4_bit;
var LCD_D5_Direction : sbit at TRISB5_bit;
var LCD_D6_Direction : sbit at TRISB6_bit;
var LCD_D7_Direction : sbit at TRISB7_bit;
// End LCD module connections

var temp : real;
    txt : array[20] of char;

// Convert ADC value to Celsius degrees format
function ADC_to_degC() : real;
  begin
    result := ADC1_Get_Sample(8);          // Read ADC value from AN8 pin
    result := (((3.25/1024) * result - 0.5) * 100);
  end;

begin
  CHECON := 0x32;
  AD1PCFG := 0xFFF7;                        // Configure AN8 pin as analog I/O
  ADC1_Init();                              // Initialize ADC
  Delay_100ms();
  Lcd_Init();                               // Initialize LCD
  Lcd_Cmd(_LCD_CLEAR);                      // Clear LCD
  Lcd_Cmd(_LCD_CURSOR_OFF);                 // Turn cursor off
  Lcd_Out(1, 1, ' Temperature:   ');

  while(TRUE) do
  begin
    temp := ADC_to_degC();              // Convert ADC value to Celsius degrees format
    FloatToStr(temp, txt);
    Lcd_Chr(2,13,223);                 // Print degree character, 'C' for Centigrades
                        // Different LCD displays have different char code for degree
    Lcd_Chr(2,14,'C');                        // If you see greek alpha letter try typing
178 instead of 223
    Lcd_Out(2, 5, txt);                     // Display value on the LCD
    Delay_1sec();                           // 1 second delay
  end;
end.
```

CANSPI Library

The SPI module is available with a number of the PIC32 MCUs. The mikroPascal PRO for PIC32 provides a library (driver) for working with mikroElektronika's CANSPI Add-on boards (with MCP2515 or MCP2510) via SPI interface.

The CAN is a very robust protocol that has error detection and signalization, self-checking and fault confinement. Faulty CAN data and remote frames are re-transmitted automatically, similar to the Ethernet.

Data transfer rates depend on distance. For example, 1 Mbit/s can be achieved at network lengths below 40m while 250 Kbit/s can be achieved at network lengths below 250m. The greater distance the lower maximum bitrate that can be achieved. The lowest bitrate defined by the standard is 200Kbit/s. Cables used are shielded twisted pairs.

CAN supports two message formats:

- Standard format, with 11 identifier bits and
- Extended format, with 29 identifier bits

Important :

- Consult the CAN standard about CAN bus termination resistance.
- An effective CANSPI communication speed depends on SPI and certainly is slower than "real" CAN.
- The library uses the SPI module for communication. User must initialize appropriate SPI module before using the SPI Graphic Lcd Library.
- For MCUs with multiple SPI modules it is possible to initialize both of them and then switch by using the SPI_Set_Active routine.
- Number of SPI modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

Library Dependency Tree

External dependencies of CANSPI Library

The following variables must be defined in all projects using CANSPI Library:	Description:	Example:
`var CanSpi_CS : sbit; sfr; external;`	Chip Select line.	`var CanSpi_CS : sbit at LATF0_bit;`
`var CanSpi_Rst : sbit; sfr; external;`	Reset line.	`var CanSpi_Rst : sbit at LATF1_bit;`
`var CanSpi_CS_Direction : sbit; sfr; external;`	Direction of the Chip Select pin.	`var CanSpi_CS_Direction : sbit at TRISF0_bit;`
`var CanSpi_Rst_Direction : sbit; sfr; external;`	Direction of the Reset pin.	`var CanSpi_Rst_Direction : sbit at TRISF1_bit;`

Library Routines

- CANSPISetOperationMode
- CANSPIGetOperationMode
- CANSPIInit
- CANSPISetBaudRate
- CANSPISetMask
- CANSPISetFilter
- CANSPIRead
- CANSPIWrite

CANSPISetOperationMode

Prototype	`procedure CANSPISetOperationMode(mode : byte; WAIT: byte);`
Description	Sets the CANSPI module to requested mode.
Parameters	- `mode`: CANSPI module operation mode. Valid values: `CANSPI_OP_MODE` constants. See `CANSPI_OP_MODE` constants. - `WAIT`: CANSPI mode switching verification request. If `WAIT == 0`, the call is non-blocking. The function does not verify if the CANSPI module is switched to requested mode or not. Caller must use `CANSPIGetOperationMode` to verify correct operation mode before performing mode specific operation. If `WAIT != 0`, the call is blocking – the function won't "return" until the requested mode is set.
Returns	Nothing.
Requires	The CANSPI routines are supported only by MCUs with the SPI module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See connection example at the bottom of this page.
Example	`// set the CANSPI module into configuration mode (wait inside CANSPISetOperationMode until this mode is set)` `CANSPISetOperationMode(_CANSPI_MODE_CONFIG, 0xFF);`
Notes	None.

CANSPIGetOperationMode

Prototype	`function CANSPIGetOperationMode() : byte;`
Description	The function returns current operation mode of the CANSPI module. Check CANSPI_OP_MODE constants or device datasheet for operation mode codes.
Parameters	None.
Returns	Current operation mode.
Requires	The CANSPI routines are supported only by MCUs with the SPI module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See connection example at the bottom of this page.

Example	`// check whether the CANSPI module is in Normal mode and if it is do something.`
	`if (CANSPIGetOperationMode() = _CANSPI_MODE_NORMAL) then`
	`begin`
	`  ...`
	`end;`

Notes	None.

CANSPIInit

Prototype	`procedure CANSPIInit(SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CANSPI_CONFIG_FLAGS : char);`
Description	Initializes the CANSPI module. Stand-Alone CAN controller in the CANSPI module is set to: - Disable CAN capture - Continue CAN operation in Idle mode - Do not abort pending transmissions - Fcan clock : 4*Tcy (Fosc) - Baud rate is set according to given parameters - CAN mode : Normal - Filter and mask registers IDs are set to zero - Filter and mask message frame type is set according to `CANSPI_CONFIG_FLAGS` value `SAM`, `SEG2PHTS`, `WAKFIL` and `DBEN` bits are set according to `CANSPI_CONFIG_FLAGS` value.
Parameters	- `SJW` as defined in MCU's datasheet (CAN Module) - `BRP` as defined in MCU's datasheet (CAN Module) - `PHSEG1` as defined in MCU's datasheet (CAN Module) - `PHSEG2` as defined in MCU's datasheet (CAN Module) - `PROPSEG` as defined in MCU's datasheet (CAN Module) - `CANSPI_CONFIG_FLAGS` is formed from predefined constants. See `CANSPI_CONFIG_FLAGS` constants.
Returns	Nothing.

<table>
<tr><td>Requires</td><td>Global variables:

- <code>CanSpi_CS</code>: Chip Select line
- <code>CanSpi_Rst</code>: Reset line
- <code>CanSpi_CS_Direction</code>: Direction of the Chip Select pin
- <code>CanSpi_Rst_Direction</code>: Direction of the Reset pin

must be defined before using this function.

The CANSPI routines are supported only by MCUs with the SPI module.

The SPI module needs to be initialized. See the <code>SPIx_Init</code> and <code>SPIx_Init_Advanced</code> routines.

MCU has to be properly connected to mikroElektronika’s CANSPI Extra Board or similar hardware. See connection example at the bottom of this page.</td></tr>
<tr><td>Example</td><td><pre>// CANSPI module connections
var CanSpi_CS   : sbit at LATF0_bit;
    CanSpi_CS_Direction : sbit at TRISF0_bit;
    CanSpi_Rst   : sbit at LATF1_bit;
    CanSpi_Rst_Direction : sbit at TRISF1_bit;
// End CANSPI module connections

var CANSPI_Init_Flags: word;
...
CANSPI_Init_Flags := _CANSPI_CONFIG_SAMPLE_THRICE and
                   _CANSPI_CONFIG_PHSEG2_PRG_ON and
                   _CANSPI_CONFIG_STD_MSG   and
                   _CANSPI_CONFIG_DBL_BUFFER_ON and
                   _CANSPI_CONFIG_VALID_XTD_MSG and
                   _CANSPI_CONFIG_LINE_FILTER_OFF;
...
SPI1_Init();   // initialize
SPI1 module
CANSPIInit(1,3,3,3,1,CANSPI_Init_Flags);   // initialize CANSPI</pre></td></tr>
<tr><td>Notes</td><td>- CANSPI mode NORMAL will be set on exit.</td></tr>
</table>

CANSPISetBaudRate

Prototype	`procedure CANSPISetBaudRate(SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CANSPI_CONFIG_FLAGS : char);`
Returns	Nothing.
Description	Sets the CANSPI module baud rate. Due to complexity of the CAN protocol, you can not simply force a bps value. Instead, use this function when the CANSPI module is in Config mode. `SAM`, `SEG2PHTS` and `WAKFIL` bits are set according to CANSPI_CONFIG_FLAGS value. Refer to datasheet for details.
Parameters	- `SJW` as defined in MCU's datasheet (CAN Module) - `BRP` as defined in MCU's datasheet (CAN Module) - `PHSEG1` as defined in MCU's datasheet (CAN Module) - `PHSEG2` as defined in MCU's datasheet (CAN Module) - `PROPSEG` as defined in MCU's datasheet (CAN Module) - `CANSPI_CONFIG_FLAGS` is formed from predefined constants. See CANSPI_CONFIG_FLAGS constants.
Returns	Nothing.
Requires	The CANSPI module must be in Config mode, otherwise the function will be ignored. See CANSPISetOperationMode. The CANSPI routines are supported only by MCUs with the SPI module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See connection example at the bottom of this page.

Example	`// set required baud rate and sampling rules`
	`var CANSPI_CONFIG_FLAGS : byte;`
	`...`
	`CANSPISetOperationMode(_CANSPI_MODE_CONFIG,0xFF);      // set CONFIGURATION mode (CANSPI module must be in config mode for baud rate settings)`
	`CANSPI_CONFIG_FLAGS := _CANSPI_CONFIG_SAMPLE_THRICE and`
	`_CANSPI_CONFIG_PHSEG2_PRG_ON   and`
	`_CANSPI_CONFIG_STD_MSG         and`
	`_CANSPI_CONFIG_DBL_BUFFER_ON   and`
	`_CANSPI_CONFIG_VALID_XTD_MSG   and`
	`_CANSPI_CONFIG_LINE_FILTER_OFF;`
	`CANSPISetBaudRate(1, 1, 3, 3, 1, CANSPI_CONFIG_FLAGS);`

Notes	None.

CANSPISetMask

Prototype	`procedure CANSPISetMask(CANSPI_MASK : byte; val : longint; CANSPI_CONFIG_FLAGS : byte);`
Description	Configures mask for advanced filtering of messages. The parameter `value` is bit-adjusted to the appropriate mask registers.

Parameters	- `CANSPI_MASK`: CAN module mask number. Valid values: `CANSPI_MASK` constants. See CANSPI_MASK constants.
	- `val`: mask register value. This value is bit-adjusted to appropriate buffer mask registers
	- `CANSPI_CONFIG_FLAGS`: selects type of message to filter. Valid values:
	  - `_CANSPI_CONFIG_ALL_VALID_MSG`,
	  - `_CANSPI_CONFIG_MATCH_MSG_TYPE & _CANSPI_CONFIG_STD_MSG`,
	  - `_CANSPI_CONFIG_MATCH_MSG_TYPE & _CANSPI_CONFIG_XTD_MSG`.
	See CANSPI_CONFIG_FLAGS constants.

Returns	Nothing.
Requires	The CANSPI module must be in Config mode, otherwise the function will be ignored. See CANSPISetOperationMode. The CANSPI routines are supported only by MCUs with the SPI module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See connection example at the bottom of this page.

Example	`// set the appropriate filter mask and message type value`
	`CANSPISetOperationMode(_CANSPI_MODE_CONFIG,0xFF); // set CONFIGURATION mode (CANSPI1 module must be in config mode for mask settings)`
	
	`// Set all B1 mask bits to 1 (all filtered bits are relevant):`
	`// Note that -1 is just a cheaper way to write 0xFFFFFFFF.`
	`// Complement will do the trick and fill it up with ones.`
	`CANSPISetMask(_CANSPI_MASK_B1, -1, _CANSPI_CONFIG_MATCH_MSG_TYPE and _CANSPI_CONFIG_XTD_MSG);`

Notes	None.

CANSPISetFilter

Prototype	`procedure CANSPISetFilter(CAN_FILTER : as byte, val : longint, CANSPI_CONFIG_FLAGS : as byte);`
Description	Configures message filter. The parameter `value` is bit-adjusted to the appropriate filter registers.
Parameters	- `CANSPI_FILTER`: CAN module filter number. Valid values: `CANSPI_FILTER` constants. See CANSPI_FILTER constants. - `val`: filter register value. This value is bit-adjusted to appropriate filter registers - `CANSPI_CONFIG_FLAGS`: selects type of message to filter. Valid values: `_CANSPI_CONFIG_STD_MSG` and `_CANSPI_CONFIG_XTD_MSG`. See CANSPI_CONFIG_FLAGS constants.
Returns	Nothing.
Requires	The CANSPI module must be in Config mode, otherwise the function will be ignored. See CANSPISetOperationMode. The CANSPI routines are supported only by MCUs with the SPI module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See connection example at the bottom of this page.

Example	`// set the appropriate filter value and message type`
	`CANSPISetOperationMode(_CANSPI_MODE_CONFIG,0xFF);    // set CONFIGURATION mode (CANSPI module must be in config mode for filter settings)`
	
	`// Set id of filter B1_F1 to 3 :`
	`CANSPISetFilter(_CANSPI_FILTER_B1_F1, 3, _CANSPI_CONFIG_XTD_MSG);`

Notes	None.

CANSPIRead

Prototype	`function CANSPIRead(var id : longint; var Data_ : array[8] of byte; var DataLen: byte; var CAN_RX_MSG_FLAGS : byte) : byte;`
Description	If at least one full Receive Buffer is found, it will be processed in the following way: - Message ID is retrieved and stored to location provided by the `id` parameter - Message data is retrieved and stored to a buffer provided by the `data` parameter - Message length is retrieved and stored to location provided by the `dataLen` parameter - Message flags are retrieved and stored to location provided by the `CANSPI_RX_MSG_FLAGS` parameter
Parameters	- `id`: message identifier address - `data`: an array of bytes up to 8 bytes in length - `dataLen`: data length address - `CANSPI_RX_MSG_FLAGS`: message flags address. For message receive flags format refer to `CANSPI_RX_MSG_FLAGS` constants. See CANSPI_RX_MSG_FLAGS constants.
Returns	- `0` if nothing is received - `0xFFFF` if one of the Receive Buffers is full (message received)
Requires	The CANSPI module must be in a mode in which receiving is possible. See CANSPISetOperationMode. The CANSPI routines are supported only by MCUs with the SPI module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See connection example at the bottom of this page.

Example	`// check the CANSPI1 module for received messages. If any was received do something.`
	`var msg_rcvd, rx_flags, data_len : byte;`
	`    data : array[8] of byte;`
	`    msg_id : longint;`
	`...`
	`CANSPISetOperationMode(_CANSPI_MODE_NORMAL,0xFF);            // set NORMAL mode (CANSPI1 module must be in mode in which receive is possible)`
	`...`
	`rx_flags := 0;                                              // clear message flags`
	`if (msg_rcvd = CANSPIRead(msg_id, data, data_len, rx_flags)) then`
	`begin`
	`  ...`
	`end;`

Notes	None.

CANSPIWrite

Prototype	`function CANSPIWrite(id : longint; var Data_ : array[8] of byte; DataLen, CANSPI_TX_MSG_FLAGS : byte) : byte;`
Description	If at least one empty Transmit Buffer is found, the function sends message in the queue for transmission.
Parameters	- `id`: CAN message identifier. Valid values: 11 or 29 bit values, depending on message type (standard or extended) - `Data`: data to be sent - `DataLen`: data length. Valid values: 0..8 - `CANSPI_TX_MSG_FLAGS`: message flags. Valid values: `CANSPI_TX_MSG_FLAGS` constants. See CANSPI_TX_MSG_FLAGS constants.
Returns	- `0` if all Transmit Buffers are busy - `0xFFFF` if at least one Transmit Buffer is available
Requires	The CANSPI module must be in mode in which transmission is possible. See CANSPISetOperationMode. The CANSPI routines are supported only by MCUs with the SPI module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See connection example at the bottom of this page.

Example	`// send message extended CAN message with the appropriate ID and data`
	`var tx_flags : byte;`
	`    data : array[8] of byte;`
	`    msg_id : longint;`
	`...`
	`CANSPISetOperationMode(CANSPI_MODE_NORMAL,0xFF);                    // set NORMAL mode (CANSPI must be in mode in which transmission is possible)`
	
	`tx_flags := _CANSPI_TX_PRIORITY_0 and _CANSPI_TX_XTD_FRAME;   // set message flags`
	`CANSPIWrite(msg_id, data, 2, tx_flags);`

Notes	None.

CANSPI Constants

There is a number of constants predefined in the CANSPI library. You need to be familiar with them in order to be able to use the library effectively. Check the example at the end of the chapter.

CANSPI_OP_MODE Constants

The `CANSPI_OP_MODE` constants define CANSPI operation mode. Function CANSPISetOperationMode expects one of these as it's argument:

Copy Code To Clipboard

```
const
    _CANSPI_MODE_BITS   : byte = $E0;   // Use this to access opmode  bits
    _CANSPI_MODE_NORMAL : byte = 0;
    _CANSPI_MODE_SLEEP  : byte = $20;
    _CANSPI_MODE_LOOP   : byte = $40;
    _CANSPI_MODE_LISTEN : byte = $60;
    _CANSPI_MODE_CONFIG : byte = $80;
```

CANSPI_CONFIG_FLAGS Constants

The `CANSPI_CONFIG_FLAGS` constants define flags related to the CANSPI module configuration. The functions CANSPIInit, CANSPISetBaudRate, CANSPISetMask and CANSPISetFilter expect one of these (or a bitwise combination) as their argument:

Copy Code To Clipboard

```
const
    _CANSPI_CONFIG_DEFAULT        : byte  = $FF;   // 11111111

    _CANSPI_CONFIG_PHSEG2_PRG_BIT : byte  = $01;
    _CANSPI_CONFIG_PHSEG2_PRG_ON  : byte  = $FF;   // XXXXXXX1
    _CANSPI_CONFIG_PHSEG2_PRG_OFF : byte  = $FE;   // XXXXXXX0

    _CANSPI_CONFIG_LINE_FILTER_BIT : byte = $02;
    _CANSPI_CONFIG_LINE_FILTER_ON  : byte = $FF;   // XXXXXX1X
    _CANSPI_CONFIG_LINE_FILTER_OFF : byte = $FD;   // XXXXXX0X

    _CANSPI_CONFIG_SAMPLE_BIT      : byte = $04;
    _CANSPI_CONFIG_SAMPLE_ONCE     : byte = $FF;   // XXXXX1XX
    _CANSPI_CONFIG_SAMPLE_THRICE   : byte = $FB;   // XXXXX0XX

    _CANSPI_CONFIG_MSG_TYPE_BIT    : byte = $08;
    _CANSPI_CONFIG_STD_MSG         : byte = $FF;   // XXXX1XXX
    _CANSPI_CONFIG_XTD_MSG         : byte = $F7;   // XXXX0XXX

    _CANSPI_CONFIG_DBL_BUFFER_BIT  : byte = $10;
    _CANSPI_CONFIG_DBL_BUFFER_ON   : byte = $FF;   // XXX1XXXX
    _CANSPI_CONFIG_DBL_BUFFER_OFF  : byte = $EF;   // XXX0XXXX

    _CANSPI_CONFIG_MSG_BITS        : byte = $60;
    _CANSPI_CONFIG_ALL_MSG         : byte = $FF;   // X11XXXXX
    _CANSPI_CONFIG_VALID_XTD_MSG   : byte = $DF;   // X10XXXXX
    _CANSPI_CONFIG_VALID_STD_MSG   : byte = $BF;   // X01XXXXX
    _CANSPI_CONFIG_ALL_VALID_MSG   : byte = $9F;   // X00XXXXX
```

You may use bitwise AND (&) to form config byte out of these values. For example:

Copy Code To Clipboard

```
init := _CANSPI_CONFIG_SAMPLE_THRICE and
       _CANSPI_CONFIG_PHSEG2_PRG_ON and
       _CANSPI_CONFIG_STD_MSG       and
       _CANSPI_CONFIG_DBL_BUFFER_ON and
       _CANSPI_CONFIG_VALID_XTD_MSG and
       _CANSPI_CONFIG_LINE_FILTER_OFF;
...
CANSPIInit(1, 1, 3, 3, 1, init);   // initialize CANSPI
```

CANSPI_TX_MSG_FLAGS Constants

`CANSPI_TX_MSG_FLAGS` are flags related to transmission of a CANSPI message:

Copy Code To Clipboard

```
const
    _CANSPI_TX_PRIORITY_BITS : byte = $03;
    _CANSPI_TX_PRIORITY_0    : byte = $FC;     // XXXXXX00
    _CANSPI_TX_PRIORITY_1    : byte = $FD;     // XXXXXX01
    _CANSPI_TX_PRIORITY_2    : byte = $FE;     // XXXXXX10
    _CANSPI_TX_PRIORITY_3    : byte = $FF;     // XXXXXX11

    _CANSPI_TX_FRAME_BIT     : byte  = $08;
    _CANSPI_TX_STD_FRAME     : byte  = $FF;     // XXXXX1XX
    _CANSPI_TX_XTD_FRAME     : byte  = $F7;     // XXXXX0XX

    _CANSPI_TX_RTR_BIT       : byte = $40;
    _CANSPI_TX_NO_RTR_FRAME  : byte = $FF;     // X1XXXXXX
    _CANSPI_TX_RTR_FRAME     : byte = $BF;     // X0XXXXXX
```

You may use bitwise AND (`and`) to adjust the appropriate flags. For example:

Copy Code To Clipboard

```
// form value to be used as sending message flag :
send_config := _CANSPI_TX_PRIORITY_0 and
               _CANSPI_TX_XTD_FRAME  and
               _CANSPI_TX_NO_RTR_FRAME;
...
CANSPIWrite(id, data, 1, send_config);
```

CANSPI_RX_MSG_FLAGS Constants

`CANSPI_RX_MSG_FLAGS` are flags related to reception of CANSPI message. If a particular bit is set then corresponding meaning is TRUE or else it will be FALSE.

Copy Code To Clipboard

```
const
    _CANSPI_RX_FILTER_BITS  : byte = $07;   // Use this to access filter bits
    _CANSPI_RX_FILTER_1     : byte = $00;
    _CANSPI_RX_FILTER_2     : byte = $01;
    _CANSPI_RX_FILTER_3     : byte = $02;
    _CANSPI_RX_FILTER_4     : byte = $03;
    _CANSPI_RX_FILTER_5     : byte = $04;
    _CANSPI_RX_FILTER_6     : byte = $05;

    _CANSPI_RX_OVERFLOW     : byte = $08;   // Set if Overflowed else cleared
    _CANSPI_RX_INVALID_MSG  : byte = $10;   // Set if invalid else cleared
    _CANSPI_RX_XTD_FRAME    : byte = $20;   // Set if XTD message else cleared
    _CANSPI_RX_RTR_FRAME    : byte = $40;  // Set if RTR message else cleared
    _CANSPI_RX_DBL_BUFFERED : byte = $80;  // Set if this message was hardware double-
buffered
```

You may use bitwise AND (`and`) to adjust the appropriate flags. For example:

Copy Code To Clipboard

```
if (MsgFlag and _CANSPI_RX_OVERFLOW) <> 0 then
begin
  ...
  // Receiver overflow has occurred.
  // We have lost our previous message.
end;
```

CANSPI_MASK Constants

The `CANSPI_MASK` constants define mask codes. Function CANSPISetMask expects one of these as it's argument:

Copy Code To Clipboard

```
const
    _CANSPI_MASK_B1 : byte = 0;
    _CANSPI_MASK_B2 : byte = 1;
```

CANSPI_FILTER Constants

The `CANSPI_FILTER` constants define filter codes. Functions CANSPISetFilter expects one of these as it's argument:

Copy Code To Clipboard

```
const
    _CANSPI_FILTER_B1_F1 : byte = 0;
    _CANSPI_FILTER_B1_F2 : byte = 1;
    _CANSPI_FILTER_B2_F1 : byte = 2;
    _CANSPI_FILTER_B2_F2 : byte = 3;
    _CANSPI_FILTER_B2_F3 : byte = 4;
    _CANSPI_FILTER_B2_F4 : byte = 5;
```

Library Example

The code is a simple demonstration of CANSPI protocol. This node initiates the communication with the 2nd node by sending some data to its address. The 2nd node responds by sending back the data incremented by 1. This (1st) node then does the same and sends incremented data back to the 2nd node, etc.

Code for the first CANSPI node:

Copy Code To Clipboard

```
program Can_Spi_1st;

const ID_1st : longint = 12111;
const ID_2nd : longint = 3;

var Can_Init_Flags, Can_Send_Flags, Can_Rcv_Flags : word;   // can flags
    Rx_Data_Len : word;                                   // received data length in bytes
    RxTx_Data   : array[8] of byte;                         // can rx/tx data buffer
    Msg_Rcvd : byte;                                        // reception flag
    Tx_ID, Rx_ID : dword;                                   // can rx and tx ID

// CANSPI module connections
var CanSpi_CS            : sbit at  LATF0_bit;
    CanSpi_CS_Direction  : sbit at TRISF0_bit;
    CanSpi_Rst           : sbit at LATF1_bit;
    CanSpi_Rst_Direction : sbit at TRISF1_bit;
// End CANSPI module connections

begin
  CHECON := 0x32;
  AD1PCFG := 0xFFFF;                              // configure AN pins as digital I/O

  PORTB := 0;                                             // clear PORTB
  TRISB := 0;                                             // set PORTB as output

  Can_Init_Flags := 0;                                    //
  Can_Send_Flags := 0;                                    // clear flags
  Can_Rcv_Flags  := 0;                                    //

  Can_Send_Flags := _CANSPI_TX_PRIORITY_0 and             // form value to be used
                    _CANSPI_TX_XTD_FRAME and              // with CANSPIWrite
                    _CANSPI_TX_NO_RTR_FRAME;

  Can_Init_Flags := _CANSPI_CONFIG_SAMPLE_THRICE and      // form value to be used
                    _CANSPI_CONFIG_PHSEG2_PRG_ON and      // with CANSPIInit
                    _CANSPI_CONFIG_XTD_MSG and
                    _CANSPI_CONFIG_DBL_BUFFER_ON and
                    _CANSPI_CONFIG_VALID_XTD_MSG;

// Initialize SPI2 module
  SPI2_Init();

  CANSPIInitialize(1,3,3,3,1,Can_Init_Flags);  // initialize external CANSPI module
  CANSPISetOperationMode(_CANSPI_MODE_CONFIG,0xFF);          // set CONFIGURATION mode
  CANSPISetMask(_CANSPI_MASK_B1,-1,_CANSPI_CONFIG_XTD_MSG); // set all mask1 bits to ones
```

```
  CANSPISetMask(_CANSPI_MASK_B2,-1,_CANSPI_CONFIG_XTD_MSG);            // set all mask2
bits to ones
   CANSPISetFilter(_CANSPI_FILTER_B2_F4,ID_2nd,_CANSPI_CONFIG_XTD_MSG);   // set id of
filter B2_F4 to 2nd node ID

  CANSPISetOperationMode(_CANSPI_MODE_NORMAL,0xFF);        // set NORMAL mode

// Set initial data to be sent
  RxTx_Data[0] := 9;

  CANSPIWrite(ID_1st, RxTx_Data, 1, Can_Send_Flags);       // send initial message

  while (TRUE) do
    begin                                                  // endless loop
     Msg_Rcvd := CANSPIRead(Rx_ID , RxTx_Data , Rx_Data_Len, Can_Rcv_Flags);   // receive
message
      if ((Rx_ID = ID_2nd) and Msg_Rcvd) then   // if message received check id
        begin
          PORTB := RxTx_Data[0];                // id correct, output data at PORTD
          Inc(RxTx_Data[0]);                    // increment received data
          Delay_ms(10);
          CANSPIWrite(ID_1st, RxTx_Data, 1, Can_Send_Flags);                 // send
incremented data back
        end;
    end;
end.
```

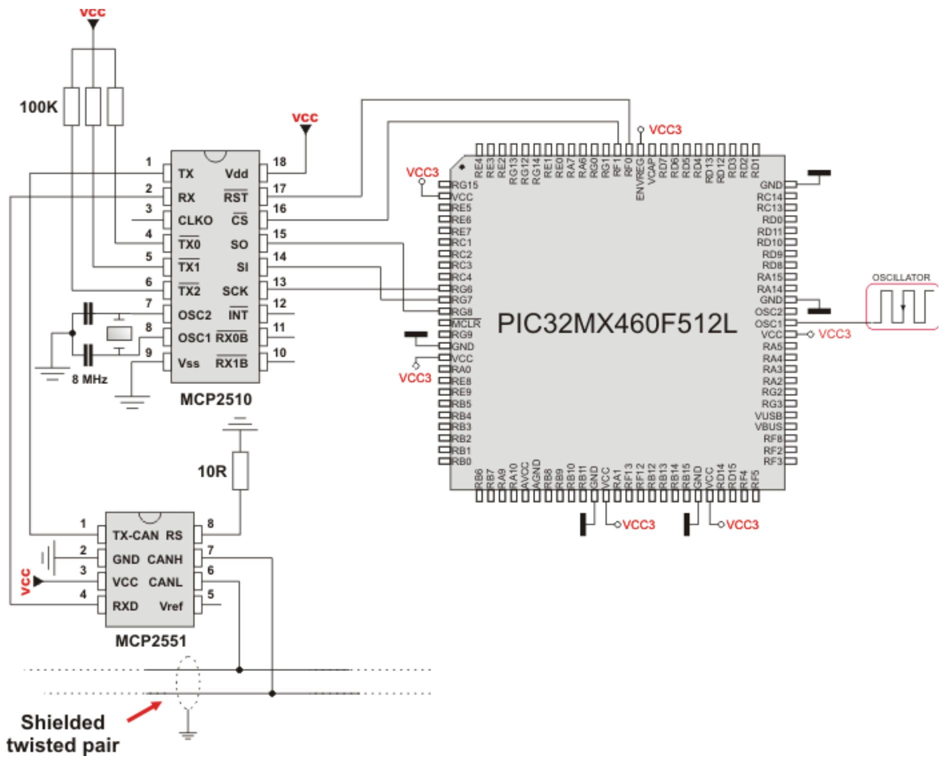
Code for the second CANSPI node:

Copy Code To Clipboard

```
program Can_Spi_2nd;

const ID_1st : longint = 12111;
const ID_2nd : longint = 3;

var Can_Init_Flags, Can_Send_Flags, Can_Rcv_Flags : word;    // can flags
    Rx_Data_Len : word;                                   // received data length in bytes
    RxTx_Data   : array[8] of byte;                       // can rx/tx data buffer
    Msg_Rcvd : byte;                                      // reception flag
    Tx_ID, Rx_ID : dword;                                 // can rx and tx ID

// CANSPI module connections
var CanSpi_CS            : sbit at  LATF0_bit;
    CanSpi_CS_Direction  : sbit at TRISF0_bit;
    CanSpi_Rst           : sbit at LATF1_bit;
    CanSpi_Rst_Direction : sbit at TRISF1_bit;
// End CANSPI module connections

begin
  CHECON := 0x32;
  AD1PCFG := 0xFFFF;                              // configure AN pins as digital I/O
```

```
  PORTB := 0;                                              // clear PORTB
  TRISB := 0;                                              // set PORTB as output

  Can_Init_Flags := 0;                                     //
  Can_Send_Flags := 0;                                     // clear flags
  Can_Rcv_Flags  := 0;                                     //

  Can_Send_Flags := _CANSPI_TX_PRIORITY_0 and              // form value to be used
                    _CANSPI_TX_XTD_FRAME and               // with CANSPIWrite
                    _CANSPI_TX_NO_RTR_FRAME;

  Can_Init_Flags := _CANSPI_CONFIG_SAMPLE_THRICE and       // form value to be used
                    _CANSPI_CONFIG_PHSEG2_PRG_ON and       // with CANSPIInit
                    _CANSPI_CONFIG_XTD_MSG and
                    _CANSPI_CONFIG_DBL_BUFFER_ON and
                    _CANSPI_CONFIG_VALID_XTD_MSG and
                    _CANSPI_CONFIG_LINE_FILTER_OFF;

// Initialize SPI1 module
  SPI2_Init();

  CANSPIInitialize(1,3,3,3,1,Can_Init_Flags);                               // initialize
external CANSPI module
  CANSPISetOperationMode(_CANSPI_MODE_CONFIG,0xFF);          // set CONFIGURATION mode
   CANSPISetMask(_CANSPI_MASK_B1,-1,_CANSPI_CONFIG_XTD_MSG);                 // set all
mask1 bits to ones
   CANSPISetMask(_CANSPI_MASK_B2,-1,_CANSPI_CONFIG_XTD_MSG);                 // set all
mask2 bits to ones
  CANSPISetFilter(_CANSPI_FILTER_B2_F3,ID_1st,_CANSPI_CONFIG_XTD_MSG);     // set id of
filter B2_F3 to 1st node ID

  CANSPISetOperationMode(_CANSPI_MODE_NORMAL,0xFF);          // set NORMAL mode

  while (TRUE) do                                            // endless loop
    begin
     Msg_Rcvd := CANSPIRead(Rx_ID , RxTx_Data , Rx_Data_Len, Can_Rcv_Flags);   // receive
message
      if ((Rx_ID = ID_1st) and Msg_Rcvd) then      // if message received check id
        begin
          PORTB := RxTx_Data[0];                   // id correct, output data at PORTB
          Inc(RxTx_Data[0]);                       // increment received data
          CANSPIWrite(ID_2nd, RxTx_Data, 1, Can_Send_Flags);                  // send
incremented data back
        end;
    end;
end.
```

HW Connection



Example of interfacing CAN transceiver MCP2510 with MCU via SPI interface

Compact Flash Library

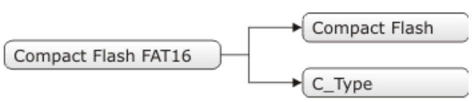
The Compact Flash Library provides routines for accessing data on Compact Flash card (abbr. CF further in text). CF cards are widely used memory elements, commonly used with digital cameras. Great capacity and excellent access time of only a few microseconds make them very attractive for microcontroller applications.

In CF card, data is divided into sectors. One sector usually comprises 512 bytes. Routines for file handling, the Cf_Fat routines, are not performed directly but successively through 512B buffer.

Important :

- Routines for file handling can be used only with FAT16 file system.
- Library functions create and read files from the root directory only.
- Library functions populate both FAT1 and FAT2 tables when writing to files, but the file data is being read from the FAT1 table only; i.e. there is no recovery if the FAT1 table gets corrupted.
- If MMC/SD card has Master Boot Record (MBR), the library will work with the first available primary (logical) partition that has non-zero size. If MMC/SD card has Volume Boot Record (i.e. there is only one logical partition and no MBRs), the library works with entire card as a single partition. For more information on MBR, physical and logical drives, primary/secondary partitions and partition tables, please consult other resources, e.g. Wikipedia and similar.
- Before writing operation, make sure not to overwrite boot or FAT sector as it could make your card on PC or digital camera unreadable. Drive mapping tools, such as Winhex, can be of great assistance.

Library Dependency Tree



External dependencies of Compact Flash Library

The following variables must be defined in all projects using Compact Flash Library:	Description:	Example:
`var CF_Data_Port : byte; sfr; external;`	Compact Flash Data Port.	`var CF_Data_Port : byte at PORTD;`
`var CF_RDY : sbit; sfr; external;`	Ready signal line.	`var CF_RDY : sbit at RB7_bit;`
`var CF_WE : sbit; sfr; external;`	Write Enable signal line.	`var CF_WE : sbit at LATB6_bit;`
`var CF_OE : sbit; sfr; external;`	Output Enable signal line.	`var CF_OE : sbit at LATB5_bit;`
`var CF_CD1 : sbit; sfr; external;`	Chip Detect signal line.	`var CF_CD1 : sbit at RB4_bit;`
`var CF_CE1 : sbit; sfr; external;`	Chip Enable signal line.	`var CF_CE1 : sbit at LATB3_bit;`
`var CF_A2 : sbit; sfr; external;`	Address pin 2.	`var CF_A2 : sbit at LATB2_bit;`
`var CF_A1 : sbit; sfr; external;`	Address pin 1.	`var CF_A1 : sbit at LATB1_bit;`
`var CF_A0 : sbit; sfr; external;`	Address pin 0.	`var CF_A0 : sbit at LATB0_bit;`
`var CF_RDY_direction : sbit; sfr; external;`	Direction of the Ready pin.	`var CF_RDY_direction : sbit at TRISB7_bit;`
`var CF_WE_direction : sbit; sfr; external;`	Direction of the Write Enable pin.	`var CF_WE_direction : sbit at TRISB6_bit;`
`var CF_OE_direction : sbit; sfr; external;`	Direction of the Output Enable pin.	`var CF_OE_direction : sbit at TRISB5_bit;`
`var CF_CD1_direction : sbit; sfr; external;`	Direction of the Chip Detect pin.	`var CF_CD1_direction : sbit at TRISB4_bit;`
`var CF_CE1_direction : sbit; sfr; external;`	Direction of the Chip Enable pin.	`var CF_CE1_direction : sbit at TRISB3_bit;`
`var CF_A2_direction : sbit; sfr; external;`	Direction of the Address 2 pin.	`var CF_A2_direction : sbit at TRISB2_bit;`
`var CF_A1_direction : sbit; sfr; external;`	Direction of the Address 1 pin.	`var CF_A1_direction : sbit at TRISB1_bit;`
`var CF_A0_direction : sbit; sfr; external;`	Direction of the Address 0 pin.	`var CF_A0_direction : sbit at TRISB0_bit;`

Library Routines

- Cf_Init
- Cf_Detect
- Cf_Enable
- Cf_Disable
- Cf_Read_Init
- Cf_Read_Byte
- Cf_Write_Init
- Cf_Write_Byte
- Cf_Read_Sector
- Cf_Write_Sector

Routines for file handling:

- Cf_Fat_Init
- Cf_Fat_QuickFormat
- Cf_Fat_Assign
- Cf_Fat_Reset
- Cf_Fat_Read
- Cf_Fat_Rewrite
- Cf_Fat_Append
- Cf_Fat_Delete
- Cf_Fat_Write
- Cf_Fat_Set_File_Date
- Cf_Fat_Get_File_Date
- Cf_Fat_Get_File_Date_Modified
- Cf_Fat_Get_File_Size
- Cf_Fat_Get_Swap_File

The following routine is for the internal use by compiler only:

- Cf_Issue_ID_Command

Cf_Init

Prototype	`procedure Cf_Init();`
Description	Initializes ports appropriately for communication with CF card.
Parameters	None.
Returns	Nothing.
Requires	Global variables: - `CF_Data_Port` : Compact Flash data port - `CF_RDY` : Ready signal line - `CF_WE` : Write enable signal line - `CF_OE` : Output enable signal line - `CF_CD1` : Chip detect signal line - `CF_CE1` : Enable signal line - `CF_A2` : Address pin 2 - `CF_A1` : Address pin 1 - `CF_A0` : Address pin 0 - `CF_RDY_direction` : Direction of the Ready pin - `CF_WE_direction` : Direction of the Write enable pin - `CF_OE_direction` : Direction of the Output enable pin - `CF_CD1_direction` : Direction of the Chip detect pin - `CF_CE1_direction` : Direction of the Chip enable pin - `CF_A2_direction` : Direction of the Address 2 pin - `CF_A1_direction` : Direction of the Address 1 pin - `CF_A0_direction` : Direction of the Address 0 pin must be defined before using this function.

Example	`// set compact flash pinout`
	`var`
	`  Cf_Data_Port : byte at PORTD;`
	
	`  CF_RDY : sbit at RB7_bit;`
	`  CF_WE  : sbit at LATB6_bit; // for writing to output pin always use latch`
	`  CF_OE  : sbit at LATB5_bit; // for writing to output pin always use latch`
	`  CF_CD1 : sbit at RB4_bit;`
	`  CF_CE1 : sbit at LATB3_bit; // for writing to output pin always use latch`
	`  CF_A2  : sbit at LATB2_bit; // for writing to output pin always use latch`
	`  CF_A1  : sbit at LATB1_bit; // for writing to output pin always use latch`
	`  CF_A0  : sbit at LATB0_bit; // for writing to output pin always use latch`
	
	`  CF_RDY_direction : sbit at TRISB7_bit;`
	`  CF_WE_direction  : sbit at TRISB6_bit;`
	`  CF_OE_direction  : sbit at TRISB5_bit;`
	`  CF_CD1_direction : sbit at TRISB4_bit;`
	`  CF_CE1_direction : sbit at TRISB3_bit;`
	`  CF_A2_direction  : sbit at TRISB2_bit;`
	`  CF_A1_direction  : sbit at TRISB1_bit;`
	`  CF_A0_direction  : sbit at TRISB0_bit;`
	`// end of compact flash pinout`
	`...`
	`Cf_Init();      // initialize CF`

Notes	None.

Cf_Detect

Prototype	`function CF_Detect() : word ;`
Description	Checks for presence of CF card by reading the `chip detect` pin.
Parameters	None.
Returns	- `1` - if CF card was detected - `0` - otherwise
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See `Cf_Init`.

Example	`// Wait until CF card is inserted:`
	`while (Cf_Detect() = 0) do`
	`  nop;`

Notes	PIC32 family MCU and CF card voltage levels are different. The user must ensure that MCU's pin connected to CD line can read CF card Logical One correctly.

Cf_Enable

Prototype	`procedure Cf_Enable();`
Description	Enables the device. Routine needs to be called only if you have disabled the device by means of the Cf_Disable routine. These two routines in conjunction allow you to free/occupy data line when working with multiple devices.
Parameters	None.
Returns	Nothing.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See Cf_Init.
Example	`// enable compact flash` `Cf_Enable();`
Notes	None.

Cf_Disable

Prototype	`procedure Cf_Disable();`
Description	Routine disables the device and frees the data lines for other devices. To enable the device again, call Cf_Enable. These two routines in conjunction allow you to free/occupy data line when working with multiple devices.
Parameters	None.
Returns	Nothing.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See Cf_Init.
Example	`// disable compact flash` `Cf_Disable();`
Notes	None.

Cf_Read_Init

Prototype	`procedure Cf_Read_Init(address : dword; sectcnt : byte);`
Description	Initializes CF card for reading.
Parameters	- `address`: the first sector to be prepared for reading operation. - `sector_count`: number of sectors to be prepared for reading operation.
Returns	Nothing.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See Cf_Init.
Example	`// initialize compact flash for reading from sector 590` `Cf_Read_Init(590, 1);`
Notes	None.

Cf_Read_Byte

Prototype	`function CF_Read_Byte() : byte;`
Description	Reads one byte from Compact Flash sector buffer location currently pointed to by internal read pointers. These pointers will be autoicremented upon reading.
Parameters	None.
Returns	Returns a byte read from Compact Flash sector buffer.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See Cf_Init. CF card must be initialized for reading operation. See Cf_Read_Init.

Example	`// Read a byte from compact flash:`
	`var data_ as byte;`
	`...`
	`data_ := Cf_Read_Byte();`

Notes	Higher byte of the `unsigned` return value is cleared.

Cf_Write_Init

Prototype	`procedure Cf_Write_Init(address : dword; sectcnt : word);`
Description	Initializes CF card for writing.
Parameters	- `address`: the first sector to be prepared for writing operation. - `sectcnt`: number of sectors to be prepared for writing operation.
Returns	Nothing.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See Cf_Init.
Example	`// initialize compact flash for writing to sector 590` `Cf_Write_Init(590, 1);`
Notes	None.

Cf_Write_Byte

Prototype	`procedure Cf_Write_Byte(data_ : byte) ;`
Description	Writes a byte to Compact Flash sector buffer location currently pointed to by writing pointers. These pointers will be autoicremented upon reading. When sector buffer is full, its contents will be transfered to appropriate flash memory sector.
Parameters	- `data_`: byte to be written.
Returns	Nothing.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See Cf_Init. CF card must be initialized for writing operation. See Cf_Write_Init.

Example	`var data_ : byte;`
	`...`
	`data_ := 0xAA;`
	`Cf_Write_Byte(data_);`

Notes	None.

Cf_Read_Sector

Prototype	`procedure Cf_Read_Sector(sector_number : dword; var buffer : array[512] of byte);`
Description	Reads one sector (512 bytes). Read data is stored into buffer provided by the `buffer` parameter.
Parameters	- `sector_number`: sector to be read. - `buffer`: data buffer of at least 512 bytes in length.
Returns	Nothing.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See Cf_Init.

Example	`// read sector 22`
	`var data_ : array[512] of byte;`
	`...`
	`Cf_Read_Sector(22, data_);`

Notes	None.

Cf_Write_Sector

Prototype	`procedure Cf_Write_Sector(sector_number : dword; var buffer : array[512] of byte) ;`
Description	Writes 512 bytes of data provided by the buffer parameter to one CF sector.
Parameters	- `sector_number`: sector to be written to. - `buffer`: data buffer of 512 bytes in length.
Returns	Nothing.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See Cf_Init.

Example	`// write to sector 22`
	`var data_ : array[512] of byte;`
	`...`
	`Cf_Write_Sector(22, data_);`

Notes	None.

Cf_Fat_Init

Prototype	`function Cf_Fat_Init(): word;`
Description	Initializes CF card, reads CF FAT16 boot sector and extracts necessary data needed by the library.
Parameters	None.
Returns	- `0` - if CF card was detected and successfully initialized - `1` - if FAT16 boot sector was not found - `255` - if card was not detected
Requires	Nothing.

Example	`//  init the FAT library`
	`if (Cf_Fat_Init() = 0) then`
	`  begin`
	`  ...`
	`  end`

Notes	None.

Cf_Fat_QuickFormat

Prototype	`function Cf_Fat_QuickFormat(var cf_fat_label : string[11]) : word;`
Description	Formats to FAT16 and initializes CF card.
Parameters	- `cf_fat_label`: volume label (11 characters in length). If less than 11 characters are provided, the label will be padded with spaces. If null string is passed, the volume will not be labeled.
Returns	- `0` - if CF card was detected, successfully formated and initialized - `1` - if FAT16 format was unsuccessful - `255` - if card was not detected
Requires	Nothing.

Example	`// format and initialize the FAT library`
	`if (Cf_Fat_QuickFormat('mikroE') = 0) then`
	`begin`
	`  ...`
	`end;`

Notes	- This routine can be used instead or in conjunction with Cf_Fat_Init routine. - If CF card already contains a valid boot sector, it will remain unchanged (except volume label field) and only FAT and ROOT tables will be erased. Also, the new volume label will be set.

Cf_Fat_Assign

<table>
<tr><td>Prototype</td><td><code>function Cf_Fat_Assign(var filename: array[12] of char; file_cre_attr: byte): word;</code></td></tr>
<tr><td>Description</td><td>Assigns file for file operations (read, write, delete...). All subsequent file operations will be applied over the assigned file.</td></tr>
<tr><td>Parameters</td><td>- <code>filename</code>: name of the file that should be assigned for file operations. The file name should be in DOS 8.3 (file_name.extension) format. The file name and extension will be automatically padded with spaces by the library if they have less than length required (i.e. “mikro.tx” -> “mikro .tx “), so the user does not have to take care of that. The file name and extension are case insensitive. The library will convert them to proper case automatically, so the user does not have to take care of that.
Also, in order to keep backward compatibility with the first version of this library, file names can be entered as UPPERCASE string of 11 bytes in length with no dot character between the file name and extension (i.e. “MIKROELETXT” -> MIKROELE.TXT). In this case the last 3 characters of the string are considered to be file extension.

- <code>file_cre_attr</code>: file creation and attributes flags. Each bit corresponds to the appropriate file attribute:

<table>
<tr><th>Bit</th><th>Mask</th><th>Description</th></tr>
<tr><td>0</td><td>0x01</td><td>Read Only</td></tr>
<tr><td>1</td><td>0x02</td><td>Hidden</td></tr>
<tr><td>2</td><td>0x04</td><td>System</td></tr>
<tr><td>3</td><td>0x08</td><td>Volume Label</td></tr>
<tr><td>4</td><td>0x10</td><td>Subdirectory</td></tr>
<tr><td>5</td><td>0x20</td><td>Archive</td></tr>
<tr><td>6</td><td>0x40</td><td>Device (internal use only, never found on disk)</td></tr>
<tr><td>7</td><td>0x80</td><td>File creation flag. If the file does not exist and this flag is set, a new file with specified name will be created.</td></tr>
</table>
</td></tr>
<tr><td>Returns</td><td>- <code>0</code> if file does not exist and no new file is created.
- <code>1</code> if file already exists or file does not exist but a new file is created.</td></tr>
<tr><td>Requires</td><td>CF card and CF library must be initialized for file operations. See Cf_Fat_Init.</td></tr>
<tr><td>Example</td><td><code>// create file with archive attributes if it does not already exist
Cf_Fat_Assign(“MIKRO007.TXT”,0xA0);</code></td></tr>
<tr><td>Notes</td><td>Long File Names (LFN) are not supported.</td></tr>
</table>

Cf_Fat_Reset

Prototype	`procedure Cf_Fat_Reset(var size: dword);`
Description	Opens currently assigned file for reading.
Parameters	- `size`: buffer to store file size to. After file has been open for reading its size is returned through this parameter.
Returns	Nothing.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign.

Example	`var size : dword;`
	`...`
	`Cf_Fat_Reset(size);`

Notes	None.

Cf_Fat_Read

Prototype	`procedure Cf_Fat_Read(var bdata: byte);`
Description	Reads a byte from currently assigned file opened for reading. Upon function execution file pointers will be set to the next character in the file.
Parameters	- `bdata`: buffer to store read byte to. Upon this function execution read byte is returned through this parameter.
Returns	Nothing.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign. File must be open for reading. See Cf_Fat_Reset.

Example	`var bdata : byte;`
	`...`
	`Cf_Fat_Read(bdata);`

Notes	None.

Cf_Fat_Rewrite

Prototype	`procedure Cf_Fat_Rewrite();`
Description	Opens currently assigned file for writing. If the file is not empty its content will be erased.
Parameters	None.
Returns	Nothing.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. The file must be previously assigned. See Cf_Fat_Assign.
Example	`// open file for writing` `Cf_Fat_Rewrite();`
Notes	None.

Cf_Fat_Append

Prototype	`procedure Cf_Fat_Append();`
Description	Opens currently assigned file for appending. Upon this function execution file pointers will be positioned after the last byte in the file, so any subsequent file writing operation will start from there.
Parameters	None.
Returns	Nothing.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign.
Example	`// open file for appending` `Cf_Fat_Append();`
Notes	None.

Cf_Fat_Delete

Prototype	`procedure Cf_Fat_Delete();`
Description	Deletes currently assigned file from CF card.
Parameters	None.
Returns	Nothing.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign.
Example	`// delete current file` `Cf_Fat_Delete();`
Notes	None.

Cf_Fat_Write

Prototype	`procedure Cf_Fat_Write(var fdata: array[512] of byte; data_len: word);`
Description	Writes requested number of bytes to currently assigned file opened for writing.
Parameters	- `fdata`: data to be written. - `data_len`: number of bytes to be written.
Returns	Nothing.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign. File must be open for writing. See Cf_Fat_Rewrite or Cf_Fat_Append.

Example	`var file_contents : array[42] of byte;`
	`...`
	`Cf_Fat_Write(file_contents, 42); // write data to the assigned file`

Notes	None.

Cf_Fat_Set_File_Date

Prototype	`procedure Cf_Fat_Set_File_Date(year: word; month: byte; day: byte; hours: byte; mins: byte; seconds: byte);`
Description	Sets the date/time stamp. Any subsequent file writing operation will write this stamp to currently assigned file's time/date attributes.
Parameters	- `year`: year attribute. Valid values: 1980-2107 - `month`: month attribute. Valid values: 1-12 - `day`: day attribute. Valid values: 1-31 - `hours`: hours attribute. Valid values: 0-23 - `mins`: minutes attribute. Valid values: 0-59 - `seconds`: seconds attribute. Valid values: 0-59
Returns	Nothing.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign. File must be open for writing. See Cf_Fat_Rewrite or Cf_Fat_Append.
Example	`Cf_Fat_Set_File_Date(2005,9,30,17,41,0);`
Notes	None.

Cf_Fat_Get_File_Date

Prototype	`procedure Cf_Fat_Get_File_Date(var year: word; var month: byte; var day: byte; var hours: byte; var mins: byte);`
Description	Reads time/date attributes of currently assigned file.
Parameters	- `year`: buffer to store year attribute to. Upon function execution year attribute is returned through this parameter. - `month`: buffer to store month attribute to. Upon function execution month attribute is returned through this parameter. - `day`: buffer to store day attribute to. Upon function execution day attribute is returned through this parameter. - `hours`: buffer to store hours attribute to. Upon function execution hours attribute is returned through this parameter. - `mins`: buffer to store minutes attribute to. Upon function execution minutes attribute is returned through this parameter.
Returns	Nothing.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign.

Example	`var year : word;`
	`    month, day, hours, mins : byte;`
	`...`
	`Cf_Fat_Get_File_Date(year, month, day, hours, mins);`

Notes	None.

Cf_Fat_Get_File_Date_Modified

Prototype	`procedure Cf_Fat_Get_File_Date_Modified(var year: word; var month: byte; var day: byte; var hours: byte; var mins: byte);`
Description	Retrieves the last modification date/time of the currently assigned file.
Parameters	- `year`: buffer to store year of modification attribute to. Upon function execution year of modification attribute is returned through this parameter. - `month`: buffer to store month of modification attribute to. Upon function execution month of modification attribute is returned through this parameter. - `day`: buffer to store day of modification attribute to. Upon function execution day of modification attribute is returned through this parameter. - `hours`: buffer to store hours of modification attribute to. Upon function execution hours of modification attribute is returned through this parameter. - `mins`: buffer to store minutes of modification attribute to. Upon function execution minutes of modification attribute is returned through this parameter.
Returns	Nothing.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign.

Example	`var year : word;`
	`    month, day, hours, mins : byte;`
	`...`
	`Cf_Fat_Get_File_Date_Modified(year, month, day, hours, mins);`

Notes	None.

Cf_Fat_Get_File_Size

Prototype	`function Cf_Fat_Get_File_Size(): dword;`
Description	This function reads size of currently assigned file in bytes.
Parameters	None.
Returns	Size of the currently assigned file in bytes.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign.

Example	`var my_file_size : dword;`
	`...`
	`my_file_size := Cf_Fat_Get_File_Size();`

Notes	None.

Cf_Fat_Get_Swap_File

Prototype	`function Cf_Fat_Get_Swap_File(sectors_cnt: dword; var filename : string[11]; file_attr : byte): dword;`
Description	This function is used to create a swap file of predefined name and size on the CF media. If a file with specified name already exists on the media, search for consecutive sectors will ignore sectors occupied by this file. Therefore, it is recommended to erase such file if it exists before calling this function. If it is not erased and there is still enough space for a new swap file, this function will delete it after allocating new memory space for a new swap file. The purpose of the swap file is to make reading and writing to CF media as fast as possible, by using the Cf_Read_Sector() and Cf_Write_Sector() functions directly, without potentially damaging the FAT system. Swap file can be considered as a “window” on the media where the user can freely write/read data. It’s main purpose in the this library is to be used for fast data acquisition; when the time-critical acquisition has finished, the data can be re-written into a “normal” file, and formatted in the most suitable way.
Parameters	- `sectors_cnt`: number of consecutive sectors that user wants the swap file to have. - `filename`: name of the file that should be assigned for file operations. The file name should be in DOS 8.3 (file_name.extension) format. The file name and extension will be automatically padded with spaces by the library if they have less than length required (i.e. “mikro.tx” -> “mikro .tx “), so the user does not have to take care of that. The file name and extension are case insensitive. The library will convert them to proper case automatically, so the user does not have to take care of that. Also, in order to keep backward compatibility with the first version of this library, file names can be entered as UPPERCASE string of 11 bytes in length with no dot character between the file name and extension (i.e. “MIKROELETXT” -> MIKROELE.TXT). In this case the last 3 characters of the string are considered to be file extension. - `file_attr`: file creation and attributes flags. Each bit corresponds to the appropriate file attribute:

<table>
<tr><td>Parameters</td><td>

Bit	Mask	Description
0	0x01	Read Only
1	0x02	Hidden
2	0x04	System
3	0x08	Volume Label
4	0x10	Subdirectory
5	0x20	Archive
6	0x40	Device (internal use only, never found on disk)
7	0x80	Not used

</td></tr>
<tr><td>Returns</td><td>- Number of the start sector for the newly created swap file, if there was enough free space on CF card to create file of required size.
- 0 - otherwise.</td></tr>
<tr><td>Requires</td><td>CF card and CF library must be initialized for file operations. See Cf_Fat_Init.</td></tr>
<tr><td>Example</td><td>

```
//  Try to create a swap file with archive atribute, whose size will be at least 1000 sectors.
//                  If it succeeds, it sends the No. of start sector over UART
var size : dword;
...
size := Cf_Fat_Get_Swap_File(1000, 'mikroE.txt', 0x20);
if (size <> 0) then
begin
  UART1_Write(0xAA);
  UART1_Write(Lo(size));
  UART1_Write(Hi(size));
  UART1_Write(Higher(size));
  UART1_Write(Highest(size));
  UART1_Write(0xAA);
end;
```

</td></tr>
<tr><td>Notes</td><td>Long File Names (LFN) are not supported.</td></tr>
</table>

Library Example

This project consists of several blocks that demonstrate various aspects of usage of the Cf_Fat16 library. These are :

- Creation of new file and writing down to it;
- Opening existing file and re-writing it (writing from start-of-file);
- Opening existing file and appending data to it (writing from end-of-file);
- Opening a file and reading data from it (sending it to USART terminal);
- Creating and modifying several files at once;
- Reading file contents;
- Deleting file(s);
- Creating the swap file (see Help for details);

Copy Code To Clipboard

```
program CF_Fat16_Test;

// set compact flash pinout
var
  Cf_Data_Port : byte at PORTE;

  CF_RDY : sbit at RD7_bit;
  CF_WE  : sbit at LATD6_bit;
  CF_OE  : sbit at LATD5_bit;
  CF_CD1 : sbit at RD4_bit;
  CF_CE1 : sbit at LATD3_bit;
  CF_A2  : sbit at LATD2_bit;
  CF_A1  : sbit at LATD1_bit;
  CF_A0  : sbit at LATD0_bit;

  CF_RDY_direction : sbit at TRISD7_bit;
  CF_WE_direction  : sbit at TRISD6_bit;
  CF_OE_direction  : sbit at TRISD5_bit;
  CF_CD1_direction : sbit at TRISD4_bit;
  CF_CE1_direction : sbit at TRISD3_bit;
  CF_A2_direction  : sbit at TRISD2_bit;
  CF_A1_direction  : sbit at TRISD1_bit;
  CF_A0_direction  : sbit at TRISD0_bit;
// end of compact flash pinout

const LINE_LEN = 39;
var
  err_txt : string[20];
  file_contents : string[LINE_LEN];

  filename : string[14];   // File names

  character : byte;
  loop, loop2 : byte;
  i, size : longint;

  Buffer : array[512] of byte;

// UART write text and new line (carriage return + line feed)
procedure UART1_Write_Line( var uart_text : string );
```

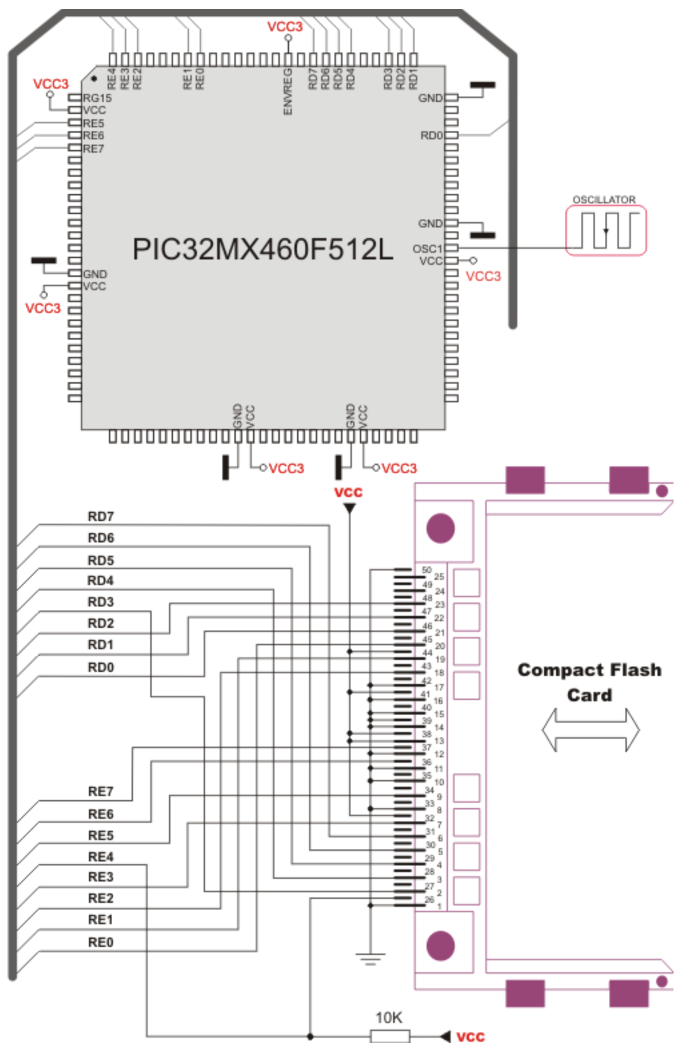
```
  begin
    UART1_Write_Text(uart_text);
    UART1_Write(13);
    UART1_Write(10);
  end;

//-------------- Creates new file and writes some data to it
procedure M_Create_New_File();
begin
  filename[7] := 'A';
  Cf_Fat_Set_File_Date(2005,6,21,10,35,0);   // Set file date & time info
  Cf_Fat_Assign(filename, 0xA0);             // Will not find file and then create file
  Cf_Fat_Rewrite();                          // To clear file and start with new data
  for loop:=1 to 99 do                       //  We want 5 files on the MMC card
    begin
      UART1_Write('.');
      file_contents[0] := loop div 10 + 48;
      file_contents[1] := loop mod 10 + 48;
      Cf_Fat_Write(file_contents, LINE_LEN-1);    // write data to the assigned file
    end;
end;

//-------------- Creates many new files and writes data to them
procedure M_Create_Multiple_Files();
begin
  for loop2 := 'B' to 'Z' do
    begin
      UART1_Write(loop2);                    // this line can slow down the performance
      filename[7] := loop2;                      // set filename
      Cf_Fat_Set_File_Date(2005,6,21,10,35,0); // Set file date & time info
      Cf_Fat_Assign(filename, 0xA0);             // find existing file or create a new one
      Cf_Fat_Rewrite();                          // To clear file and start with new data
      for loop := 1 to 44 do
        begin
          file_contents[0] := loop div 10 + 48;
          file_contents[1] := loop mod 10 + 48;
          Cf_Fat_Write(file_contents, LINE_LEN-1); // write data to the assigned file
        end;
    end;
end;

//-------------- Opens an existing file and rewrites it
procedure M_Open_File_Rewrite();
begin
  filename[7] := 'C';                           // Set filename for single-file tests
  Cf_Fat_Assign(filename, 0);
  Cf_Fat_Rewrite();
  for loop := 1 to 55 do
   begin
      file_contents[0] := byte(loop div 10 + 48);
      file_contents[1] := byte(loop mod 10 + 48);
      Cf_Fat_Write(file_contents, LINE_LEN-1);     // write data to the assigned file
    end;
end;
```

```
//-------------- Opens an existing file and appends data to it
//               (and alters the date/time stamp)
procedure M_Open_File_Append();
 begin
   filename[7] := 'B';
   Cf_Fat_Assign(filename, 0);
   Cf_Fat_Set_File_Date(2009, 1, 23, 17, 22, 0);
   Cf_Fat_Append;
   file_contents := ' for mikroElektronika 2009';  // Prepare file for append
   file_contents[26] := 13;                        // CR
   file_contents[27] := 10;                        // LF
   Cf_Fat_Write(file_contents, 27);                // Write data to assigned file
 end;

//-------------- Opens an existing file, reads data from it and puts it to USART
procedure M_Open_File_Read();
begin
  filename[7] := 'B';
  Cf_Fat_Assign(filename, 0);
  Cf_Fat_Reset(size);             // To read file, procedure returns size of file
  while size > 0 do
    begin
      Cf_Fat_Read(character);
      UART1_Write(character);     // Write data to USART
      Dec(size);
    end;
end;

//-------------- Deletes a file. If file doesn't exist, it will first be created
//               and then deleted.
procedure M_Delete_File();
begin
  filename[7] := 'F';
  Cf_Fat_Assign(filename, 0);
  Cf_Fat_Delete();
end;

//-------------- Tests whether file exists, and if so sends its creation date
//               and file size via USART
procedure M_Test_File_Exist();
var
  fsize : longint;
  year : word;
  month, day, hour, minute : byte;
  outstr : array[12] of char;
begin
 filename[7] := 'B';       // uncomment this line to search for file that DOES exists
//  filename[7] := 'F';    // uncomment this line to search for file that DOES NOT exist
  if Cf_Fat_Assign(filename, 0) <> 0 then
    begin
      //--- file has been found - get its date
      Cf_Fat_Get_File_Date(year,month,day,hour,minute);
      UART1_Write_Text(' created: ');
      WordToStr(year, outstr);
      UART1_Write_Text(outstr);
      ByteToStr(month, outstr);
```

```
      UART1_Write_Text(outstr);
      WordToStr(day, outstr);
      UART1_Write_Text(outstr);
      WordToStr(hour, outstr);
      UART1_Write_Text(outstr);
      WordToStr(minute, outstr);
      UART1_Write_Text(outstr);

    //--- file has been found - get its modified date
      Cf_Fat_Get_File_Date_Modified(year, month, day, hour, minute);
      UART1_Write_Text(' modified: ');
      WordToStr(year, outstr);
      UART1_Write_Text(outstr);
      ByteToStr(month, outstr);
      UART1_Write_Text(outstr);
      WordToStr(day, outstr);
      UART1_Write_Text(outstr);
      WordToStr(hour, outstr);
      UART1_Write_Text(outstr);
      WordToStr(minute, outstr);
      UART1_Write_Text(outstr);

      //--- get file size
      fsize := Cf_Fat_Get_File_Size;
      LongIntToStr(fsize, outstr);
      UART1_Write_Line(outstr);
    end
  else begin
    //--- file was not found - signal it
    UART1_Write(0x55);
    Delay_ms(1000);
    UART1_Write(0x55);
  end;
end;

//-------------- Tries to create a swap file, whose size will be at least 100
//                sectors (see Help for details)
procedure M_Create_Swap_File();
  var i : word;

  begin
    for i:=0 to 511 do
      Buffer[i] := i;

    size := Cf_Fat_Get_Swap_File(5000, 'mikroE.txt', 0x20);   // see help on this
function for details

    if (size <> 0) then
     begin
        LongIntToStr(size, err_txt);
        UART1_Write_Line(err_txt);
        for i:=0 to 4999 do
          begin
            Cf_Write_Sector(size, Buffer);
```

```
            Inc(size);
            UART1_Write('.');
          end;
      end;
  end;

//-------------- Main. Uncomment the function(s) to test the desired operation(s)
begin
  err_txt := 'FAT16 not found';
  file_contents := 'XX CF FAT16 library by Anton Rieckert';
  file_contents[LINE_LEN-2] := 13;
  file_contents[LINE_LEN-1] := 10;
  file_contents[LINE_LEN] := 0;
  filename := 'MIKRO00xTXT';

  {$define COMPLETE_EXAMPLE}      // comment this line to make simpler/smaller example
  CHECON := 0x32;
  AD1PCFG := 0xFFFF;              // disable A/D inputs

  // Initialize UART1 module
  UART1_Init(56000);
  Delay_ms(10);

  UART1_Write_Line('MCU-Started');          // MCU present report

  // --- Init the FAT library
  // --- use Cf_Fat_QuickFormat instead of init routine if a format is needed
  if Cf_Fat_Init() = 0 then
   begin
     Delay_ms(2000);                // wait for a while until the card is stabilized
                                    // period depends on used CF card
     //--- Test start
     UART1_Write_Line('Test Start.');
     M_Create_New_File();
     {$IFDEF COMPLETE_EXAMPLE}
     M_Create_Multiple_Files();
     M_Open_File_Rewrite();
     M_Open_File_Append();
     M_Open_File_Read();
     M_Delete_File();
     M_Test_File_Exist();
     M_Create_Swap_File();
     {$ENDIF}
     UART1_Write_Line('Test End.');
     end
   else
     begin
        UART1_Write_Line(err_txt);     // Note: Cf_Fat_Init tries to initialize a card
more than once.
                                    //       If card is not present, initialization may
last longer (depending on clock speed)
     end;
end.
```

HW Connection



Pin diagram of CF memory card

Epson S1D13700 Graphic Lcd Library

The mikroPascal PRO for PIC32 provides a library for working with Glcds based on Epson S1D13700 controller.

The S1D13700 Glcd is capable of displaying both text and graphics on an LCD panel. The S1D13700 Glcd allows layered text and graphics, scrolling of the display in any direction, and partitioning of the display into multiple screens. It includes 32K bytes of embedded SRAM display memory which is used to store text, character codes, and bit-mapped graphics.

The S1D13700 Glcd handles display controller functions including :

- Transferring data from the controlling microprocessor to the buffer memory
- Reading memory data, converting data to display pixels
- Generating timing signals for the LCD panel

The S1D13700 Glcd is designed with an internal character generator which supports 160, 5x7 pixel characters in internal mask ROM (CGROM) and 64, 8x8 pixel characters incharacter generator RAM (CGRAM).
When the CGROM is not used, up to 256, 8x16 pixel characters are supported in CGRAM.

External dependencies of the Epson S1D13700 Graphic Lcd Library

The following variables must be defined in all projects using S1D13700 Graphic Lcd library:	Description:	Example:
`var S1D13700_DATA : byte; sfr; external;`	System data bus.	`var S1D13700_DATA at PORTD;`
`var S1D13700_WR : sbit; sfr; external;`	Write signal.	`var S1D13700_WR : sbit at LATC2_bit;`
`var S1D13700_RD : sbit; sfr; external;`	Read signal.	`var S1D13700_RD : sbit at LATC1_bit;`
`var S1D13700_A0 : sbit; sfr; external;`	System Address pin.	`var S1D13700_A0 : sbit at LATC0_bit;`
`var S1D13700_RES : sbit; sfr; external;`	Reset signal.	`var S1D13700_RES : sbit at LATC3_bit;`
`var S1D13700_CS : sbit; sfr; external;`	Chip select.	`var S1D13700_CS : sbit at LATC4_bit;`
`var S1D13700_DATA_Direction : byte; sfr; external;`	Direction of the system data bus pins.	`var S1D13700_DATA_Direction sbit at PORTD;`
`var S1D13700_WR_Direction : sbit; sfr; external;`	Direction of the Write pin.	`var S1D13700_WR_Direction : sbit at TRISC2_bit;`
`var S1D13700_RD_Direction : sbit; sfr; external;`	Direction of the Read pin.	`var S1D13700_RD_Direction : sbit at TRISC1_bit;`
`var S1D13700_A0_Direction : sbit; sfr; external;`	Direction of the System Address pin.	`var S1D13700_A0_Direction : sbit at TRISC0_bit;`
`var S1D13700_RES_Direction : sbit; sfr; external;`	Direction of the Reset pin.	`var S1D13700_RES_Direction : sbit at TRISC3_bit;`
`var S1D13700_CS_Direction : sbit; sfr; external;`	Direction of the Chip select pin.	`var S1D13700_CS_Direction : sbit at TRISC4_bit;`

Library Routines

- S1D13700_Init
- S1D13700_Write_Command
- S1D13700_Write_Parameter
- S1D13700_Read_Parameter
- S1D13700_Fill
- S1D13700_GrFill
- S1D13700_TxtFill
- S1D13700_Display_GrLayer
- S1D13700_Display_TxtLayer
- S1D13700_Set_Cursor
- S1D13700_Display_Cursor
- S1D13700_Write_Char
- S1D13700_Write_Text
- S1D13700_Dot
- S1D13700_Line
- S1D13700_H_Line
- S1D13700_V_Line
- S1D13700_Rectangle
- S1D13700_Box
- S1D13700_Rectangle_Round_Edges
- S1D13700_Rectangle_Round_Edges_Fill
- S1D13700_Circle
- S1D13700_Circle_Fill
- S1D13700_Image
- S1D13700_PartialImage

S1D13700_Init

Prototype	`procedure S1D13700_Init(width : word; height : word);`
Returns	Nothing.
Description	Initializes S1D13700 Graphic Lcd controller. Parameters: - `width`: width of the Glcd panel. - `height`: height of the Glcd panel.
Requires	Global variables: - `S1D13700_Data_Port`: Data Bus Port. - `S1D13700_WR`: Write signal pin. - `S1D13700_RD`: Read signal pin. - `S1D13700_A0`: Command/Data signal pin. - `S1D13700_RES`: Reset signal pin. - `S1D13700_CS`: Chip Select signal pin. - `S1D13700_Data_Port_Direction`: Data Bus Port Direction. - `S1D13700_WR_Direction`: Direction of Write signal pin. - `S1D13700_RD_Direction`: Direction of Read signal pin. - `S1D13700_A0_Direction`: Direction of Command/Data signal pin. - `S1D13700_RES_Direction`: Direction of Reset signal pin. - `S1D13700_CS_Direction`: Direction of Chip Select signal pin. must be defined before using this function.
Example	see below

```
// S1D13700 module connections
var S1D13700_Data_Port : byte at PORTD;
var S1D13700_WR : sbit at LATC2_bit;
var S1D13700_RD : sbit at LATC1_bit;
var S1D13700_A0 : sbit at LATC0_bit;
var S1D13700_RES : sbit at LATC3_bit;
var S1D13700_CS : sbit at LATC4_bit;

var S1D13700_Data_Port_Direction : byte at PORTD;
var S1D13700_WR_Direction : sbit at TRISC2_bit;
var S1D13700_RD_Direction : sbit at TRISC1_bit;
var S1D13700_A0_Direction : sbit at TRISC0_bit;
var S1D13700_RES_Direction : sbit at TRISC3_bit;
var S1D13700_CS_Direction : sbit at TRISC4_bit;
// End of S1D13700 module connections
...
// init display for 320 pixel width, 240 pixel height
S1D13700_Init(320, 240);
```

S1D13700_Write_Command

Prototype	`procedure S1D13700_Write_Command(command : byte);`
Returns	Nothing.
Description	Writes a command to S1D13700 controller. Parameters: - `command`: command to be issued:
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`// Turn the display on` `S1D13700_Write_Command(S1D13700_DISP_ON);`

Value	Description
S1D13700_SYSTEM_SET	General system settings.
S1D13700_POWER_SAVE	Enter into power saving mode.
S1D13700_DISP_ON	Turn the display on.
S1D13700_DISP_OFF	Turn the display off.
S1D13700_SCROLL	Setup text and graphics address regions.
S1D13700_CS_RIGHT	Cursor moves right after write to display memory.
S1D13700_CS_LEFT	Cursor moves left after write to display memory.
S1D13700_CS_UP	Cursor moves up after write to display memory.
S1D13700_CS_DOWN	Cursor moves down after write to display memory.
S1D13700_OVLAY	Configure how layers overlay.
S1D13700_CGRAM_ADR	Configure character generator RAM address.
S1D13700_HDOT_SCR	Set horizontal scroll rate.
S1D13700_CSRW	Set the cursor address.
S1D13700_CSRR	Read the cursor address.
S1D13700_GRAYSCALE	Selects the gray scale depth, in bits-per-pixel (bpp).
S1D13700_MEMWRITE	Write to display memory.
S1D13700_MEMREAD	Read from display memory.

S1D13700_Write_Parameter

Prototype	`procedure S1D13700_Write_Parameter(parameter : byte);`
Returns	Nothing.
Description	Writes a parameter to S1D13700 controller. Parameters: - `parameter`: parameter to be written.
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine. Previously, a command must be sent through S1D13700_Write_Command routine.

Example	`S1D13700_Write_Command(S1D13700_CSRW); // set cursor address`
	`S1D13700_Write_Parameter(Lo(start)); // send lower byte of cursor address`
	`S1D13700_Write_Parameter(Hi(start)); // send higher byte cursor address`

S1D13700_Read_Parameter

Prototype	`function S1D13700_Read_Parameter() : byte;`
Returns	Nothing.
Description	Reads a parameter from GLCD port.
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`parameter = S1D13700_Read_Parameter();`

S1D13700_Fill

Prototype	`procedure S1D13700_Fill(d : byte; start : word; len : word);`
Returns	Nothing.
Description	Fills Glcd memory block with given byte. Parameters: - `d`: byte to be written. - `start`: starting address of the memory block. - `len`: length of the memory block in bytes.
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`// from the starting address of 0x3000, fill the memory block size of 0x7FFF with 0x20` `S1D13700_Fill(0x20, 0x3000, 0x7FFF);`

S1D13700_GrFill

Prototype	`procedure S1D13700_GrFill(d : byte);`
Returns	Nothing.
Description	Fill graphic layer with appropriate value (0 to clear). Parameters: - `d`: value to fill graphic layer with.
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`// clear current graphic panel` `S1D13700_GrFill(0);`

S1D13700_TxtFill

Prototype	`procedure S1D13700_TxtFill(d : byte);`
Returns	Nothing.
Description	Fill current text panel with appropriate value (0 to clear). Parameters: - `d`: this value will be used to fill text panel.
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`// clear current text panel` `S1D13700_TxtFill(0);`

S1D13700_Display_GrLayer

Prototype	`procedure S1D13700_Display_GrLayer(mode : byte);`
Returns	Nothing.
Description	Display selected graphic layer. Parameters: - `mode`: graphic layer mode. Valid values: (see table below)
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`// Turn on graphic layer` `S1D13700_Display_GrLayer(S1D13700_LAYER_ON);`

Value	Description
`S1D13700_LAYER_OFF`	Turn off graphic layer.
`S1D13700_LAYER_ON`	Turn on graphic layer.
`S1D13700_LAYER_FLASH_2Hz`	Turn on graphic layer and flash it at the rate of 2 Hz.
`S1D13700_LAYER_FLASH_16Hz`	Turn on graphic layer and flash it at the rate of 16 Hz.

S1D13700_Display_TxtLayer

Prototype	`procedure S1D13700_Display_TxtLayer(mode : byte);`
Returns	Nothing.
Description	Display selected text layer. Parameters: - `mode`: text layer mode. Valid values: <table><tr><th>Value</th><th>Description</th></tr><tr><td>S1D13700_LAYER_OFF</td><td>Turn off graphic layer.</td></tr><tr><td>S1D13700_LAYER_ON</td><td>Turn on graphic layer.</td></tr><tr><td>S1D13700_LAYER_FLASH_2Hz</td><td>Turn on graphic layer and flash it at the rate of 2 Hz.</td></tr><tr><td>S1D13700_LAYER_FLASH_16Hz</td><td>Turn on graphic layer and flash it at the rate of 16 Hz.</td></tr></table>
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`// Display on text layer` `S1D13700_Display_TxtLayer(S1D13700_LAYER_ON);`

S1D13700_Set_Cursor

Prototype	`procedure S1D13700_Set_Cursor(width : byte; height : byte; mode : byte);`
Returns	Nothing.
Description	Sets cursor properties. Parameters: - `width`: in pixels-1 (must be less than or equal to the horizontal char size). - `height`: in lines-1 (must be less than or equal to the vertical char size). - `mode`: cursor mode. Valid values: <table><tr><th>Value</th><th>Description</th></tr><tr><td>S1D13700_CURSOR_UNDERSCORE</td><td>Set cursor shape - underscore.</td></tr><tr><td>S1D13700_CURSOR_BLOCK</td><td>Set cursor shape - block.</td></tr></table>
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`// set cursor with the following properties : width 5px, height 10px, cursor shape - block` `S1D13700_Set_Cursor(5, 10, S1D13700_CURSOR_BLOCK);`

S1D13700_Display_Cursor

Prototype	`procedure S1D13700_Display_Cursor(mode : byte);`
Returns	Nothing.
Description	Displays cursor. Parameters: - `mode`: mode parameter. Valid values: <table><tr><th>Value</th><th>Description</th></tr><tr><td>S1D13700_CURSOR_OFF</td><td>Turn off graphic layer.</td></tr><tr><td>S1D13700_CURSOR_ON</td><td>Turn on graphic layer.</td></tr><tr><td>S1D13700_CURSOR_FLASH_2Hz</td><td>Turn on graphic layer and flash it at the rate of 2 Hz.</td></tr><tr><td>S1D13700_CURSOR_FLASH_16Hz</td><td>Turn on graphic layer and flash it at the rate of 16 Hz.</td></tr></table>
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`// set cursor on` `S1D13700_Display_Cursor(S1D13700_CURSOR_ON);`

S1D13700_Write_Char

Prototype	`procedure S1D13700_Write_Char(c : char; x : word; y : word; mode: byte);`
Returns	Nothing.
Description	Writes a char in the current text layer of Glcd at coordinates (`x, y`). Parameters: - `c`: char to be written. - `x`: char position on x-axis (column). - `y`: char position on y-axis (row). - `mode`: mode parameter. Valid values : <table><tr><th>Value</th><th>Description</th></tr><tr><td>S1D13700_OVERLAY_OR</td><td>In the OR-Mode, text and graphics can be displayed and the data is logically “OR-ed”. This is the most common way of combining text and graphics, for example labels on buttons.</td></tr><tr><td>S1D13700_OVERLAY_XOR</td><td>In this mode, the text and graphics data are combined via the logical “exclusive OR”.</td></tr><tr><td>S1D13700_OVERLAY_AND</td><td>The text and graphic data shown on display are combined via the logical “AND function”.</td></tr></table>
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`S1D13700_Write_Char('A',22,23,S1D13700_OVERLAY_OR);`

S1D13700_Write_Text

Prototype	`procedure S1D13700_Write_Text(var str : string; x, y : word; mode : byte);`
Returns	Nothing.
Description	Writes text in the current text panel of Glcd at coordinates (`x, y`). Parameters: - `str`: text to be written. - `x`: text position on x-axis (column). - `y`: text position on y-axis (row). - `mode`: mode parameter. Valid values : **Value** / **Description** `S1D13700_OVERLAY_OR` — In the OR-Mode, text and graphics can be displayed and the data is logically "OR-ed". This is the most common way of combining text and graphics, for example labels on buttons. `S1D13700_OVERLAY_XOR` — In this mode, the text and graphics data are combined via the logical "exclusive OR". `S1D13700_OVERLAY_AND` — The text and graphic data shown on display are combined via the logical "AND function".
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`S1D13700_Write_Text('EPSON LIBRARY DEMO, WELCOME !', 0, 0, S1D13700_OVERLAY_OR);`

Value	Description
`S1D13700_OVERLAY_OR`	In the OR-Mode, text and graphics can be displayed and the data is logically "OR-ed". This is the most common way of combining text and graphics, for example labels on buttons.
`S1D13700_OVERLAY_XOR`	In this mode, the text and graphics data are combined via the logical "exclusive OR".
`S1D13700_OVERLAY_AND`	The text and graphic data shown on display are combined via the logical "AND function".

S1D13700_Dot

Prototype	`procedure S1D13700_Dot(x : word; y : word; color : byte);`
Returns	Nothing.
Description	Draws a dot in the current graphic panel of Glcd at coordinates (x, y). Parameters: - `x`: dot position on x-axis. - `y`: dot position on y-axis. - `color`: color parameter. Valid values :
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`S1D13700_Dot(50, 50, S1D13700_WHITE);`

Value	Description
`S1D13700_BLACK`	Black color.
`S1D13700_WHITE`	White color.

S1D13700_Line

Prototype	`procedure S1D13700_Line(x0, y0, x1, y1 : word; pcolor : byte);`
Returns	Nothing.
Description	Draws a line from (x0, y0) to (x1, y1). Parameters: - `x0`: x coordinate of the line start. - `y0`: y coordinate of the line end. - `x1`: x coordinate of the line start. - `y1`: y coordinate of the line end. - `pcolor`: color parameter. Valid values: **Value** / **Description** `S1D13700_BLACK` – Black color. `S1D13700_WHITE` – White color.
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`S1D13700_Line(0, 0, 239, 127, S1D13700_WHITE);`

S1D13700_H_Line

Prototype	`procedure S1D13700_H_Line(x_start, x_end, y_pos : word; color : byte);`
Returns	Nothing.
Description	Draws a horizontal line. Parameters: - `x_start`: `x` coordinate of the line start. - `x_end`: `x` coordinate of the line end. - `y_pos`: line position on the y axis. - `pcolor`: color parameter. Valid values : **Value** / **Description** `S1D13700_BLACK` – Black color. `S1D13700_WHITE` – White color.
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`S1D13700_Line(0, 0, 239, 127, S1D13700_WHITE);`

S1D13700_V_Line

Prototype	`procedure S1D13700_V_Line(y_start, y_end, x_pos : word; color : byte);`
Returns	Nothing.
Description	Draws a horizontal line. Parameters: - `y_start`: y coordinate of the line start. - `y_end`: y coordinate of the line end. - `x_pos`: line position on the x axis. - `pcolor`: color parameter. Valid values: **Value** / **Description** `S1D13700_BLACK` – Black color. `S1D13700_WHITE` – White color.
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`S1D13700_Line(0, 0, 239, 127, S1D13700_WHITE);`

S1D13700_Rectangle

Prototype	`procedure S1D13700_Rectangle(x0, y0, x1, y1 : word; pcolor : byte);`
Returns	Nothing.
Description	Draws a rectangle on Glcd. Parameters: - `x0`: x coordinate of the upper left rectangle corner. - `y0`: y coordinate of the upper left rectangle corner. - `x1`: x coordinate of the lower right rectangle corner. - `y1`: y coordinate of the lower right rectangle corner. - `pcolor`: color parameter. Valid values : **Value** / **Description** `S1D13700_BLACK` – Black color. `S1D13700_WHITE` – White color.
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`S1D13700_rectangle(20, 20, 219, 107, S1D13700_WHITE);`

S1D13700_Box

Prototype	`procedure S1D13700_Box(x0, y0, x1, y1 : word; pcolor : byte);`
Returns	Nothing.
Description	Draws a rectangle on Glcd. Parameters: - `x0`: x coordinate of the upper left rectangle corner. - `y0`: y coordinate of the upper left rectangle corner. - `x1`: x coordinate of the lower right rectangle corner. - `y1`: y coordinate of the lower right rectangle corner. - `pcolor`: color parameter. Valid values : **Value** / **Description** `S1D13700_BLACK` — Black color. `S1D13700_WHITE` — White color.
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`S1D13700_Box(0, 119, 239, 127, S1D13700_WHITE);`

S1D13700_Rectangle_Round_Edges

Prototype	`procedure S1D13700_Rectangle_Round_Edges(x_upper_left : word; y_upper_left : word; x_bottom_right : word; y_bottom_right : word; round_radius : word; color : byte);`
Returns	Nothing.
Description	Draws a rounded edge rectangle on Glcd. Parameters: - `x_upper_left`: x coordinate of the upper left rectangle corner. - `y_upper_left`: y coordinate of the upper left rectangle corner. - `x_bottom_right`: x coordinate of the lower right rectangle corner. - `y_bottom_right`: y coordinate of the lower right rectangle corner. - `round_radius`: radius of the rounded edge. - `pcolor`: color parameter. Valid values: **Value** / **Description** `S1D13700_BLACK` — Black color. `S1D13700_WHITE` — White color.
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`S1D13700_Rectangle_Round_Edges(20, 20, 219, 107, 12, S1D13700_WHITE);`

S1D13700_Rectangle_Round_Edges_Fill

Prototype	`procedure S1D13700_Rectangle_Round_Edges_Fill(x_upper_left : word; y_upper_left : word; x_bottom_right : word; y_bottom_right : word; round_radius : word; color : byte);`
Returns	Nothing.
Description	Draws a filled rounded edge rectangle on Glcd. Parameters: - `x_upper_left`: x coordinate of the upper left rectangle corner. - `y_upper_left`: y coordinate of the upper left rectangle corner. - `x_bottom_right`: x coordinate of the lower right rectangle corner. - `y_bottom_right`: y coordinate of the lower right rectangle corner. - `round_radius`: radius of the rounded edge. - `pcolor`: color parameter. Valid values : **Value** / **Description** `S1D13700_BLACK` — Black color. `S1D13700_WHITE` — White color.
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`S1D13700_Rectangle_Round_Edges_Fill(20, 20, 219, 107, 12, S1D13700_WHITE);`

S1D13700_Circle

Prototype	`procedure S1D13700_Circle(x_center : word; y_center : word; radius : word; color : byte);`
Returns	Nothing.
Description	Draws a circle on Glcd. Parameters: - `x_center`: x coordinate of the circle center. - `y_center`: y coordinate of the circle center. - `radius`: radius size. - `color`: color parameter. Valid values : **Value** / **Description** `S1D13700_BLACK` — Black color. `S1D13700_WHITE` — White color.
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`S1D13700_Circle(120, 64, 110, S1D13700_WHITE);`

S1D13700_Circle_Fill

Prototype	`procedure S1D13700_Circle_Fill(x_center: word; y_center: word; radius: word; color : byte);`
Returns	Nothing.
Description	Draws a filled circle on Glcd. Parameters: - `x_center`: x coordinate of the circle center. - `y_center`: y coordinate of the circle center. - `radius`: radius size. - `color`: color parameter. Valid values : **Value** / **Description**: `S1D13700_BLACK` — Black color. `S1D13700_WHITE` — White color.
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`S1D13700_Circle_Fill(120, 64, 110, S1D13700_WHITE);`

Value	Description
`S1D13700_BLACK`	Black color.
`S1D13700_WHITE`	White color.

S1D13700_Image

Prototype	`procedure S1D13700_Image(const image : ^byte);`
Returns	Nothing.
Description	Displays bitmap on Glcd. Parameters: - `image`: image to be displayed. Bitmap array is located in code memory. **Note**: Image dimension must match the display dimension.
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`S1D13700_Image(image);`

S1D13700_PartialImage

Prototype	`procedure S1D13700_PartialImage(x_left, y_top, width, height, picture_width, picture_height : word; const image : ^byte);`
Returns	Nothing.
Description	Displays a partial area of the image on a desired location. Parameters: - `x_left`: x coordinate of the desired location (upper left coordinate). - `y_top`: y coordinate of the desired location (upper left coordinate). - `width`: desired image width. - `height`: desired image height. - `picture_width`: width of the original image. - `picture_height`: height of the original image. - `image`: image to be displayed. Bitmap array is located in code memory. **Note** : Image dimension must match the display dimension.
Requires	Glcd module needs to be initialized. See the S1D13700_Init routine.
Example	`// Draws a 10x15 part of the image starting from the upper left corner on the coordinate (10,12). Original image size is 16x32.` `S1D13700_PartialImage(10, 12, 10, 15, 16, 32, image);`

Flash Memory Library

This library provides routines for accessing microcontroller's (internal) Flash memory.

The program Flash array for the PIC32MX device is built up of a series of rows. A row contains 128 32-bit instruction words or 512 bytes. A group of 8 rows compose a page; which, therefore, contains 8 512 = 4096 bytes or 1024 instruction words.

A page of Flash is the smallest unit of memory that can be erased at a single time. The program Flash array can be programmed in one of two ways:

- Row programming, with 128 instruction words at a time.
- Word programming, with 1 instruction word at a time.

The CPU stalls (waits) until the programming operation is finished. The CPU will not execute any instruction, or respond to interrupts, during this time. If any interrupts occur during the programming cycle, they remain pending until the cycle completes.

Library Routines

- Flash_Write_Word
- Flash_Write_Row
- Flash_Erase_Page

Flash_Write_Word

Prototype	`procedure FLASH_Erase32(flash_address : longint);`
Description	Writes one 32-bit word in the program Flash memory on the designated address.
Parameters	- `address`: address of the FLASH memory word - `wdata`: data to be written
Returns	Nothing.
Requires	Nothing.
Example	
Notes	None.

Flash_Write_Row

Prototype	`procedure FLASH_Write_Block(flash_address : longint; data_address : word);`
Description	Writes one row in the program Flash memory (128 32-bit words or 512 bytes) on the designated address.
Parameters	- `address`: address of the FLASH memory word - `rdata`: data to be written
Returns	Nothing.
Requires	Nothing.
Example	
Notes	None.

Flash_Erase_Page

Prototype	`procedure FLASH_Write_Init(flash_address : longint; data_address : word);`
Description	Erases one page (8 rows, 1024 32-bit word, 4096 bytes) from the program Flash memory.
Parameters	- `address`: starting address of the FLASH memory block
Returns	Nothing.
Requires	Nothing.
Example	
Notes	None.

Graphic Lcd Library

mikroPascal PRO for PIC32 provides a library for operating Graphic Lcd 128x64 (with commonly used Samsung KS108/KS107 controller).

For creating a custom set of Glcd images use Glcd Bitmap Editor Tool.

Library Dependency Tree

External dependencies of Graphic Lcd Library

The following variables must be defined in all projects using Graphic Lcd Library:	Description:	Example:
`var GLCD_D0 : sbit; sfr; external;`	Data 0 line.	`var GLCD_D0 : sbit at RB0_bit;`
`var GLCD_D1 : sbit; sfr; external;`	Data 1 line.	`var GLCD_D1 : sbit at RB1_bit;`
`var GLCD_D2 : sbit; sfr; external;`	Data 2 line.	`var GLCD_D2 : sbit at RB2_bit;`
`var GLCD_D3 : sbit; sfr; external;`	Data 3 line.	`var GLCD_D3 : sbit at RB3_bit;`
`var GLCD_D4 : sbit; sfr; external;`	Data 4 line.	`var GLCD_D4 : sbit at RD0_bit;`
`var GLCD_D5 : sbit; sfr; external;`	Data 5 line.	`var GLCD_D5 : sbit at RD1_bit;`
`var GLCD_D6 : sbit; sfr; external;`	Data 6 line.	`var GLCD_D6 : sbit at RD2_bit;`
`var GLCD_D7 : sbit; sfr; external;`	Data 7 line.	`var GLCD_D7 : sbit at RD3_bit;`
`var GLCD_CS1 : sbit; sfr; external;`	Chip Select 1 line.	`var GLCD_CS1 : sbit at LATB4_bit;`
`var GLCD_CS2 : sbit; sfr; external;`	Chip Select 2 line.	`var GLCD_CS2 : sbit at LATB5_bit;`
`var GLCD_RS : sbit; sfr; external;`	Register select line.	`var GLCD_RS : sbit at LATF0_bit;`
`var GLCD_RW : sbit; sfr; external;`	Read/Write line.	`var GLCD_RW : sbit at LATF1_bit;`
`var GLCD_EN : sbit; sfr; external;`	Enable line.	`var GLCD_EN : sbit at LATF4_bit;`
`var GLCD_RST : sbit; sfr; external;`	Reset line.	`var GLCD_RST : sbit at LATF5_bit;`
`var GLCD_D0_Direction : sbit; sfr; external;`	Direction of the Data 0 pin.	`var GLCD_D0_Direction : sbit at TRISB0_bit;`
`var GLCD_D1_Direction : sbit; sfr; external;`	Direction of the Data 1 pin.	`var GLCD_D1_Direction : sbit at TRISB1_bit;`
`var GLCD_D2_Direction : sbit; sfr; external;`	Direction of the Data 2 pin.	`var GLCD_D2_Direction : sbit at TRISB2_bit;`
`var GLCD_D3_Direction : sbit; sfr; external;`	Direction of the Data 3 pin.	`var GLCD_D3_Direction : sbit at TRISB3_bit;`
`var GLCD_D4_Direction : sbit; sfr; external;`	Direction of the Data 4 pin.	`var GLCD_D4_Direction : sbit at TRISD0_bit;`
`var GLCD_D5_Direction : sbit; sfr; external;`	Direction of the Data 5 pin.	`var GLCD_D5_Direction : sbit at TRISD1_bit;`
`var GLCD_D6_Direction : sbit; sfr; external;`	Direction of the Data 6 pin.	`var GLCD_D6_Direction : sbit at TRISD2_bit;`
`var GLCD_D7_Direction : sbit; sfr; external;`	Direction of the Data 7 pin.	`var GLCD_D7_Direction : sbit at TRISD3_bit;`
`var GLCD_CS1_Direction : sbit; sfr; external;`	Direction of the Chip Select 1 pin.	`var GLCD_CS1_Direction : sbit at TRISB4_bit;`
`var GLCD_CS2_Direction : sbit; sfr; external;`	Direction of the Chip Select 2 pin.	`var GLCD_CS2_Direction : sbit at TRISB5_bit;`
`var GLCD_RS_Direction : sbit; sfr; external;`	Direction of the Register select pin.	`var GLCD_RS_Direction : sbit at TRISF0_bit;`
`var GLCD_RW_Direction : sbit; sfr; external;`	Direction of the Read/Write pin.	`var GLCD_RW_Direction : sbit at TRISF1_bit;`
`var GLCD_EN_Direction : sbit; sfr; external;`	Direction of the Enable pin.	`var GLCD_EN_Direction : sbit at TRISF4_bit;`
`var GLCD_RST_Direction : sbit; sfr; external;`	Direction of the Reset pin.	`var GLCD_RST_Direction : sbit at TRISF5_bit;`

Library Routines

Basic routines:

- Glcd_Init
- Glcd_Set_Side
- Glcd_Set_X
- Glcd_Set_Page
- Glcd_Read_Data
- Glcd_Write_Data

Advanced routines:

- Glcd_Fill
- Glcd_Dot
- Glcd_Line
- Glcd_V_Line
- Glcd_H_Line
- Glcd_Rectangle
- Glcd_Rectangle_Round_Edges
- Glcd_Rectangle_Round_Edges_Fill
- Glcd_Box
- Glcd_Circle
- Glcd_Circle_Fill
- Glcd_Set_Font
- Glcd_Write_Char
- Glcd_Write_Text
- Glcd_Image
- Glcd_PartialImage

Glcd_Init

Prototype	`procedure Glcd_Init();`
Description	Initializes the Glcd module. Each of the control lines are both port and pin configurable, while data lines must be on a single port (pins <0:7>).
Parameters	None.
Returns	Nothing.
Requires	Global variables: - `GLCD_D0` : Data pin 0 - `GLCD_D1` : Data pin 1 - `GLCD_D2` : Data pin 2 - `GLCD_D3` : Data pin 3 - `GLCD_D4` : Data pin 4 - `GLCD_D5` : Data pin 5 - `GLCD_D6` : Data pin 6 - `GLCD_D7` : Data pin 7 - `GLCD_CS1` : Chip select 1 signal pin - `GLCD_CS2` : Chip select 2 signal pin - `GLCD_RS` : Register select signal pin - `GLCD_RW` : Read/Write Signal pin

Requires	- `GLCD_EN` : Enable signal pin - `GLCD_RST` : Reset signal pin - `GLCD_D0_Direction` : Direction of the Data pin 0 - `GLCD_D1_Direction` : Direction of the Data pin 1 - `GLCD_D2_Direction` : Direction of the Data pin 2 - `GLCD_D3_Direction` : Direction of the Data pin 3 - `GLCD_D4_Direction` : Direction of the Data pin 4 - `GLCD_D5_Direction` : Direction of the Data pin 5 - `GLCD_D6_Direction` : Direction of the Data pin 6 - `GLCD_D7_Direction` : Direction of the Data pin 7 - `GLCD_CS1_Direction` : Direction of the Chip select 1 pin - `GLCD_CS2_Direction` : Direction of the Chip select 2 pin - `GLCD_RS_Direction` : Direction of the Register select signal pin - `GLCD_RW_Direction` : Direction of the Read/Write signal pin - `GLCD_EN_Direction` : Direction of the Enable signal pin - `GLCD_RST_Direction` : Direction of the Reset signal pin must be defined before using this function.
Example	see code below
Notes	None.

Example

```
// Glcd module connections
var GLCD_D7 : sbit at RD3_bit;
    GLCD_D6 : sbit at RD2_bit;
    GLCD_D5 : sbit at RD1_bit;
    GLCD_D4 : sbit at RD0_bit;
    GLCD_D3 : sbit at RB3_bit;
    GLCD_D2 : sbit at RB2_bit;
    GLCD_D1 : sbit at RB1_bit;
    GLCD_D0 : sbit at RB0_bit;
    GLCD_D7_Direction : sbit at TRISD3_bit;
    GLCD_D6_Direction : sbit at TRISD2_bit;
    GLCD_D5_Direction : sbit at TRISD1_bit;
    GLCD_D4_Direction : sbit at TRISD0_bit;
    GLCD_D3_Direction : sbit at TRISB3_bit;
    GLCD_D2_Direction : sbit at TRISB2_bit;
    GLCD_D1_Direction : sbit at TRISB1_bit;
    GLCD_D0_Direction : sbit at TRISB0_bit;
    GLCD_CS2 : sbit at LATB5_bit;
    GLCD_RS  : sbit at LATF0_bit;
    GLCD_RW  : sbit at LATF1_bit;
    GLCD_EN  : sbit at LATF4_bit;
    GLCD_RST : sbit at LATF5_bit;

var GLCD_CS1_Direction : sbit at TRISB4_bit;
    GLCD_CS2_Direction : sbit at TRISB5_bit;
    GLCD_RS_Direction  : sbit at TRISF0_bit;
    GLCD_RW_Direction  : sbit at TRISF1_bit;
    GLCD_EN_Direction  : sbit at TRISF4_bit;
    GLCD_RST_Direction : sbit at TRISF5_bit;
// End Glcd module connections
...
Glcd_Init();
```

Glcd_Set_Side

Prototype	`procedure Glcd_Set_Side(x_pos: byte);`
Description	Selects Glcd side. Refer to the Glcd datasheet for detailed explanation.
Parameters	- `x_pos`: Specifies position on x-axis of the Glcd. Valid values: 0..127. Values from 0 to 63 specify the left side, values from 64 to 127 specify the right side of the Glcd.
Returns	Nothing.
Requires	Glcd needs to be initialized, see Glcd_Init routine.

Example	The following two lines are equivalent, and both of them select the left side of Glcd:
	
	`Glcd_Select_Side(0);`
	`Glcd_Select_Side(10);`

Notes	For side, x axis and page layout explanation see schematic at the bottom of this page.

Glcd_Set_X

Prototype	`procedure Glcd_Set_X(x_pos: byte);`
Description	Sets x-axis position to `x_pos` dots from the left border of Glcd within the selected side.
Parameters	- `x_pos`: position on x-axis. Valid values: 0..63
Returns	Nothing.
Requires	Glcd needs to be initialized, see Glcd_Init routine.
Example	`Glcd_Set_X(25);`
Notes	For side, x axis and page layout explanation see schematic at the bottom of this page.

Glcd_Set_Page

Prototype	`procedure Glcd_Set_Page(page: byte);`
Description	Selects page of the Glcd.
Parameters	- `page`: page number. Valid values: 0..7
Returns	Nothing.
Requires	Glcd needs to be initialized, see Glcd_Init routine.
Example	`Glcd_Set_Page(5);`
Notes	For side, x axis and page layout explanation see schematic at the bottom of this page.

Glcd_Read_Data

Prototype	`function Glcd_Read_Data() : byte;`
Description	Reads data from from the current location of Glcd memory and moves to the next location.
Parameters	None.
Returns	One byte from Glcd memory, formatted as a word (16-bit).
Requires	Glcd needs to be initialized, see Glcd_Init routine. Glcd side, x-axis position and page should be set first. See functions Glcd_Set_Side, Glcd_Set_X, and Glcd_Set_Page.

Example	`var data_ : byte;`
	`...`
	`Glcd_Read_Data();`
	`data_ := Glcd_Read_Data();`

Notes	This routine needs to be called twice; After the first call, data is placed in the buffer register. After the second call, data is passed from the buffer register to data lines.

Glcd_Write_Data

Prototype	`procedure Glcd_Write_Data(data_: byte);`
Returns	Nothing.
Description	Writes one byte to the current location in Glcd memory and moves to the next location. Parameters: - `data_`: data to be written
Requires	Glcd needs to be initialized, see Glcd_Init routine. Glcd side, x-axis position and page should be set first. See functions Glcd_Set_Side, Glcd_Set_X, and Glcd_Set_Page.

Example	`var data_ : byte;`
	`...`
	`Glcd_Write_Data(data_);`

Glcd_Fill

Prototype	`procedure Glcd_Fill(pattern: byte);`
Description	Fills Glcd memory with the byte pattern. To clear the Glcd screen, use `Glcd_Fill(0)`. To fill the screen completely, use `Glcd_Fill(0xFF)`.
Parameters	- `pattern`: byte to fill Glcd memory with.
Returns	Nothing.
Requires	Glcd needs to be initialized, see Glcd_Init routine.
Example	`// Clear screen` `Glcd_Fill(0);`
Notes	None.

Glcd_Dot

Prototype	`procedure Glcd_Dot(x_pos, y_pos, color: byte);`
Description	Draws a dot on Glcd at coordinates (`x_pos, y_pos`).
Parameters	- `x_pos`: x position. Valid values: 0..127 - `y_pos`: y position. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter `color` determines a dot state: 0 clears dot, 1 puts a dot, and 2 inverts dot state.
Returns	Nothing.
Requires	Glcd needs to be initialized, see Glcd_Init routine.
Example	`// Invert the dot in the upper left corner` `Glcd_Dot(0, 0, 2);`
Notes	For x and y axis layout explanation see schematic at the bottom of this page.

Glcd_Line

Prototype	`procedure Glcd_Line(x_start, y_start, x_end, y_end: integer; color: byte);`
Description	Draws a line on Glcd.
Parameters	- `x_start`: x coordinate of the line start. Valid values: 0..127 - `y_start`: y coordinate of the line start. Valid values: 0..63 - `x_end`: x coordinate of the line end. Valid values: 0..127 - `y_end`: y coordinate of the line end. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the line color: 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized, see Glcd_Init routine.
Example	`// Draw a line between dots (0,0) and (20,30)` `Glcd_Line(0, 0, 20, 30, 1);`
Notes	None.

Glcd_V_Line

Prototype	`procedure Glcd_V_Line(y_start, y_end, x_pos, color: byte);`
Description	Draws a vertical line on Glcd.
Parameters	- `y_start`: y coordinate of the line start. Valid values: 0..63 - `y_end`: y coordinate of the line end. Valid values: 0..63 - `x_pos`: x coordinate of vertical line. Valid values: 0..127 - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the line color: 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized, see Glcd_Init routine.
Example	`// Draw a vertical line between dots (10,5) and (10,25)` `Glcd_V_Line(5, 25, 10, 1);`
Notes	None.

Glcd_H_Line

Prototype	`procedure Glcd_H_Line(x_start, x_end, y_pos, color: byte);`
Description	Draws a horizontal line on Glcd.
Parameters	- `x_start`: x coordinate of the line start. Valid values: 0..127 - `x_end`: x coordinate of the line end. Valid values: 0..127 - `y_pos`: y coordinate of horizontal line. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the line color: 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized, see Glcd_Init routine.
Example	`// Draw a horizontal line between dots (10,20) and (50,20)` `Glcd_H_Line(10, 50, 20, 1);`
Notes	None.

Glcd_Rectangle

Prototype	`procedure Glcd_Rectangle(x_upper_left, y_upper_left, x_bottom_right, y_bottom_right, color: byte);`
Description	Draws a rectangle on Glcd.
Parameters	- `x_upper_left`: x coordinate of the upper left rectangle corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left rectangle corner. Valid values: 0..63 - `x_bottom_right`: x coordinate of the lower right rectangle corner. Valid values: 0..127 - `y_bottom_right`: y coordinate of the lower right rectangle corner. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the color of the rectangle border: 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized, see Glcd_Init routine.
Example	`// Draw a rectangle between dots (5,5) and (40,40)` `Glcd_Rectangle(5, 5, 40, 40, 1);`
Notes	None.

Glcd_Rectangle_Round_Edges

Prototype	`procedure Glcd_Rectangle_Round_Edges(x_upper_left: byte; y_upper_left: byte; x_bottom_right: byte; y_bottom_right: byte; radius: byte; color: byte);`
Description	Draws a rounded edge rectangle on Glcd.
Parameters	- `x_upper_left`: x coordinate of the upper left rectangle corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left rectangle corner. Valid values: 0..63 - `x_bottom_right`: x coordinate of the lower right rectangle corner. Valid values: 0..127 - `y_bottom_right`: y coordinate of the lower right rectangle corner. Valid values: 0..63 - `round_radius`: radius of the rounded edge. - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the color of the rectangle border: 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized, see Glcd_Init routine.
Example	`// Draw a rounded edge rectangle between dots (5,5) and (40,40) with the radius of 12` `Glcd_Rectangle_Round_Edges(5, 5, 40, 40, 12, 1);`
Notes	None.

Glcd_Rectangle_Round_Edges_Fill

Prototype	`procedure Glcd_Rectangle_Round_Edges_Fill(x_upper_left: byte; y_upper_left: byte; x_bottom_right: byte; y_bottom_right: byte; radius: byte; color: byte);`
Description	Draws a filled rounded edge rectangle on Glcd with color.
Parameters	- `x_upper_left`: x coordinate of the upper left rectangle corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left rectangle corner. Valid values: 0..63 - `x_bottom_right`: x coordinate of the lower right rectangle corner. Valid values: 0..127 - `y_bottom_right`: y coordinate of the lower right rectangle corner. Valid values: 0..63 - `round_radius`: radius of the rounded edge - `color`: color parameter. Valid values: 0..2 The parameter color determines the `color` of the rectangle border: 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized, see Glcd_Init routine.
Example	`// Draws a filled rounded edge rectangle between dots (5,5) and (40,40) with the radius of 12` `Glcd_Rectangle_Round_Edges_Fill(5, 5, 40, 40, 12, 1);`
Notes	None.

Glcd_Box

Prototype	`procedure Glcd_Box(x_upper_left, y_upper_left, x_bottom_right, y_bottom_right, color: byte);`
Description	Draws a box on Glcd. Parameters:
Parameters	- `x_upper_left`: x coordinate of the upper left box corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left box corner. Valid values: 0..63 - `x_bottom_right`: x coordinate of the lower right box corner. Valid values: 0..127 - `y_bottom_right`: y coordinate of the lower right box corner. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the color of the box fill: 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized, see Glcd_Init routine.
Example	`// Draw a box between dots (5,15) and (20,40)` `Glcd_Box(5, 15, 20, 40, 1);`
Notes	None.

Glcd_Circle

Prototype	`procedure Glcd_Circle(x_center, y_center, radius: integer; color: byte);`
Description	Draws a circle on Glcd.
Parameters	- `x_center`: x coordinate of the circle center. Valid values: 0..127 - `y_center`: y coordinate of the circle center. Valid values: 0..63 - `radius`: radius size - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the color of the circle line: 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized, see Glcd_Init routine.
Example	`// Draw a circle with center in (50,50) and radius=10` `Glcd_Circle(50, 50, 10, 1);`
Notes	None.

Glcd_Circle_Fill

Prototype	`procedure Glcd_Circle_Fill(x_center: integer; y_center: integer; radius: integer; color: byte);`
Description	Draws a filled circle on Glcd.
Parameters	- `x_center`: x coordinate of the circle center. Valid values: 0..127 - `y_center`: y coordinate of the circle center. Valid values: 0..63 - `radius`: radius size - `color`: color parameter. Valid values: 0..2
Returns	Nothing.
Requires	Glcd needs to be initialized, see Glcd_Init routine.
Example	`// Draws a filled circle with center in (50,50) and radius=10` `Glcd_Circle_Fill(50, 50, 10, 1);`
Notes	None.

Glcd_Set_Font

Prototype	`procedure Glcd_Set_Font(const activeFont: ^byte; aFontWidth, aFontHeight : byte; aFontOffs : byte);`
Description	Sets font that will be used with Glcd_Write_Char and Glcd_Write_Text routines.
Parameters	- `activeFont`: font to be set. Needs to be formatted as an array of char - `aFontWidth`: width of the font characters in dots. - `aFontHeight`: height of the font characters in dots. - `aFontOffs`: number that represents difference between the mikroPascal PRO for PIC32 character set and regular ASCII set (eg. if 'A' is 65 in ASCII character, and 'A' is 45 in the mikroPascal PRO for PIC32 character set, aFontOffs is 20). Demo fonts supplied with the library have an offset of 32, which means that they start with space. The user can use fonts given in the file "__Lib_GLCDFonts" file located in the Uses folder or create his own fonts. List of supported fonts: - `Font_Glcd_System3x5` - `Font_Glcd_System5x7` - `Font_Glcd_5x7` - `Font_Glcd_Character8x7` For the sake of the backward compatibility, these fonts are supported also: - `System3x5` (equivalent to `Font_Glcd_System3x5`) - `FontSystem5x7_v2` (equivalent to `Font_Glcd_System5x7`) - `font5x7` (equivalent to `Font_Glcd_5x7`) - `Character8x7` (equivalent to `Font_Glcd_Character8x7`)
Returns	Nothing.
Requires	Glcd needs to be initialized, see Glcd_Init routine.
Example	`// Use the custom 5x7 font "myfont" which starts with space (32):` `Glcd_Set_Font(&myfont, 5, 7, 32);`
Notes	None.

Glcd_Write_Char

Prototype	`procedure Glcd_Write_Char(character, x_pos, page_num, color : byte);`
Description	Prints character on the Glcd.
Parameters	- `character`: character to be written - `x_pos`: character starting position on x-axis. Valid values: 0..(127-FontWidth) - `page_num`: the number of the page on which character will be written. Valid values: 0..7 - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the color of the character: 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized, see Glcd_Init routine. Use Glcd_Set_Font to specify the font for display; if no font is specified, then default `Font_Glcd_System5x7` font supplied with the library will be used.
Example	`// Write character 'C' on the position 10 inside the page 2:` `Glcd_Write_Char('C', 10, 2, 1);`
Notes	For x axis and page layout explanation see schematic at the bottom of this page.

Glcd_Write_Text

Prototype	`procedure Glcd_Write_Text(var text: string; x_pos, page_num, color : byte);`
Description	Prints text on Glcd.
Parameters	- `text`: text to be written - `x_pos`: text starting position on x-axis. - `page_num`: the number of the page on which text will be written. Valid values: 0..7 - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the color of the text: 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized, see Glcd_Init routine. Use Glcd_Set_Font to specify the font for display; if no font is specified, then default `Font_Glcd_System5x7` font supplied with the library will be used.
Example	`// Write text "Hello world!" on the position 10 inside the page 2:` `Glcd_Write_Text("Hello world!", 10, 2, 1);`
Notes	For x axis and page layout explanation see schematic at the bottom of this page.

Glcd_Image

Prototype	`procedure Glcd_Image(const image: ^byte);`
Description	Displays bitmap on Glcd.
Parameters	- `image`: image to be displayed. Bitmap array can be located in both code and RAM memory (due to the mikroPascal PRO for PIC32 pointer to const and pointer to RAM equivalency).
Returns	Nothing.
Requires	Glcd needs to be initialized, see Glcd_Init routine.
Example	`// Draw image my_image on Glcd` `Glcd_Image(my_image);`
Notes	Use the mikroPascal PRO for PIC32 integrated Glcd Bitmap Editor, Tools > Glcd Bitmap Editor, to convert image to a constant array suitable for displaying on Glcd.

Glcd_PartialImage

Prototype	`procedure Glcd_PartialImage(x_left, y_top, width, height, picture_width, picture_height : word; const image : ^byte);`
Description	Displays a partial area of the image on a desired location.
Parameters	- `x_left`: x coordinate of the desired location (upper left coordinate). - `y_top`: y coordinate of the desired location (upper left coordinate). - `width`: desired image width. - `height`: desired image height. - `picture_width`: width of the original image. - `picture_height`: height of the original image. - `image`: image to be displayed. Bitmap array can be located in both code and RAM memory (due to the mikroPascal PRO for PIC pointer to const and pointer to RAM equivalency).
Returns	Nothing.
Requires	Glcd needs to be initialized, see Glcd_Init routine.
Example	`// Draws a 10x15 part of the image starting from the upper left corner on the coordinate (10,12). Original image size is 16x32.` `Glcd_PartialImage(10, 12, 10, 15, 16, 32, image);`
Notes	Use the mikroPascal PRO for PIC32 integrated Glcd Bitmap Editor, **Tools > Glcd Bitmap Editor**, to convert image to a constant array suitable for displaying on Glcd.

I²C Library

The I²C full master I²C module is available with a number of the PIC32 MCU models. The mikroPascal PRO for PIC32 provides a library which supports the master I²C mode.

Important :

- I²C library routines require you to specify the module you want to use. To select the desired I²C module, simply change the letter x in the routine prototype for a number from **1** to **3**.
- Number of I²C modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

Library Routines

- I2Cx_Init
- I2Cx_Init_Advanced
- I2Cx_Start
- I2Cx_Restart
- I2Cx_Is_Idle
- I2Cx_Read
- I2Cx_Write
- I2Cx_Stop

I2Cx_Init

Prototype	`procedure I2Cx_Init(scl : longint);`
Description	This function configures and initializes the desired I²C module with default settings. This function enables the I²C module by setting the I2CEN bit. The rest of the bits in I²C control register remains unchanged. Default initialization (after reset) of I²C module is: - Continue operation in IDLE mode. - 7-bit slave address. - Slew rate control enabled for High Speed mode (400 kHz). - General call address disabled. - SCL clock stretching disabled. As per the I²C standard, SCL clock may be 100 kHz or 400 kHz. However, the user can specify any clock rate up to 1 MHz.
Parameters	- `scl`: requested serial clock rate.
Returns	Nothing.
Requires	MCU with the I²C module.
Example	`// Initialize the I2C1 module with clock_rate of 100000` `I2C1_Init(100000);`
Notes	Refer to the MCU's datasheet for correct values of the scl in respect with Fosc. I²C library routines require you to specify the module you want to use. To select the desired I²C module, simply change the letter **x** in the routine prototype for a number from **1** to **3**. Number of I²C modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

I2Cx_Init_Advanced

Prototype	`procedure I2Cx_Init_Advanced(Fclk_Khz, scl : dword);`
Description	This function configures and initializes the desired I²C module using Peripheral Bus Clock and default initialization settings. As per the I²C standard, SCL clock may be 100 kHz or 400 kHz. However, the user can specify any clock rate up to 1 MHz.
Parameters	- `Fclk_Khz`: Peripheral Bus Clock frequency in kHz. - `scl`: requested serial clock rate.
Returns	Nothing.
Requires	MCU with the I²C module.
Example	
Notes	- I²C library routines require you to specify the module you want to use. To select the desired I²C module, simply change the letter **x** in the routine prototype for a number from **1** to **5**. - Number of I²C modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

I2Cx_Start

Prototype	`procedure I2Cx_Start();`
Description	Determines if the I²C bus is free and issues START signal.
Parameters	None.
Returns	Nothing.
Requires	MCU with at least one I²C module. Used I²C module must be initialized before using this function. See I2Cx_Init routine.
Example	`// Issue START signal` `I2C1_Start();`
Notes	- I²C library routines require you to specify the module you want to use. To select the desired I²C module, simply change the letter **x** in the routine prototype for a number from **1** to **5**. - Number of I²C modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

I2Cx_Restart

Prototype	`procedure I2Cx_Restart();`
Description	Issues repeated START signal.
Parameters	None.
Returns	Nothing.
Requires	MCU with at least one I²C module. Used I²C module must be initialized before using this function. See I2Cx_Init routine.
Example	`// Issue RESTART signal` `I2C1_Restart();`
Notes	- I²C library routines require you to specify the module you want to use. To select the desired I²C module, simply change the letter **x** in the routine prototype for a number from **1** to **5**. - Number of I²C modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

I2Cx_Is_Idle

Prototype	`function I2Cx_Is_Idle() : word;`
Description	Waits for the I²C bus to become free. This is a blocking function.
Parameters	None.
Returns	- `0` if I²C bus is free. - `1` if I²C bus is not free.
Requires	MCU with at least one I²C module. Used I²C module must be initialized before using this function. See I2Cx_Init routine.

Example	`var data_ : byte;`
	`...`
	`if !(I2C1_Is_Idle)`
	`  I2C1_Write(data_);`
	`...`

Notes	- I²C library routines require you to specify the module you want to use. To select the desired I²C module, simply change the letter **x** in the routine prototype for a number from **1** to **5**. - Number of I²C modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

I2Cx_Read

Prototype	`function I2Cx_Read(ack : word) : byte;`
Description	Reads a byte from the I²C bus.
Parameters	- `ack`: acknowledge signal parameter. If the `ack = 0`, *acknowledge* signal will be sent after reading, otherwise the *not acknowledge* signal will be sent.
Returns	Received data.
Requires	MCU with at least one I²C module. Used I²C module must be initialized before using this function. See I2Cx_Init routine. Also, START signal needs to be issued in order to use this function. See I2Cx_Start.

Example	`var take : byte;`
	`...`
	`// Read data and send the not_acknowledge signal`
	`take := I2C1_Read(1);`

Notes	- I²C library routines require you to specify the module you want to use. To select the desired I²C module, simply change the letter **x** in the routine prototype for a number from **1** to **5**. - Number of I²C modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

I2Cx_Write

Prototype	`function I2Cx_Write(data_ : byte) : word;`
Description	Sends data byte via the I²C bus.
Parameters	- `data_`: data to be sent
Returns	- `0` if there were no errors. - `1` if write collision was detected on the I²C bus.
Requires	MCU with at least one I²C module. Used I²C module must be initialized before using this function. See I2Cx_Init routine. Also, START signal needs to be issued in order to use this function. See I2Cx_Start.

Example	`var data_ : byte;`
	`    error : word;`
	`...`
	`error := I2C1_Write(data_);`
	`error := I2C1_Write(0xA3);`

Notes	- I²C library routines require you to specify the module you want to use. To select the desired I²C module, simply change the letter **x** in the routine prototype for a number from **1** to **5**. - Number of I²C modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

I2Cx_Stop

Prototype	`procedure I2Cx_Stop();`
Description	Issues STOP signal.
Parameters	None.
Returns	Nothing.
Requires	MCU with at least one I²C module. Used I²C module must be initialized before using this function. See I2Cx_Init routine.
Example	`// Issue STOP signal` `I2C1_Stop();`
Notes	- I²C library routines require you to specify the module you want to use. To select the desired I²C module, simply change the letter **x** in the routine prototype for a number from **1** to **5**. - Number of I²C modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

Library Example

This code demonstrates working with the I²C library. Program sends data to EEPROM (data is written at the address 2). After that, program reads data from the same EEPROM address and displays it on PORTB for visual check. See the figure below how to interface the 24C02 to PIC32.

Copy Code To Clipboard

```
program I2C_Simple;

var i, b : char;

procedure EEPROM_24C02_Init();
begin
  I2C2_Init(100000);
end;

//--------------- Writes data to 24C02 EEPROM - signle location
procedure EEPROM_24C02_WrSingle(wAddr : byte; wData : byte);
  begin
    I2C2_Start();           // issue I2C start signal
    I2C2_Write(0xA0);       // send byte via I2C  (command to 24cO2)
    I2C2_Write(wAddr);      // send byte (address of EEPROM location)
    I2C2_Write(wData);      // send data (data to be written)
    I2C2_Stop();
  end;

//--------------- Reads data from 24C02 EEPROM - single location (random)
function EEPROM_24C02_RdSingle(rAddr : byte) : byte;
  begin
    I2C2_Start();              // issue I2C start signal
    I2C2_Write(0xA0);          // send byte via I2C  (device address + W)
    I2C2_Write(rAddr);         // send byte (data address)
    I2C2_Restart();            // issue I2C signal repeated start
    I2C2_Write(0xA1);          // send byte (device address + R)
```

```
    result := I2C2_Read(1);      // Read the data (NO acknowledge)
    I2C2_Stop();
  end;

begin
  CHECON := 0x30;
  AD1PCFG := 0xFFFFFFFF;

  LATB := 0;                   // Set PORTB value to zero
  TRISB := 0;                  // Configure PORTB as output
  TRISA := 0;
  TRISD := 0;
  LATD := 0;
  LATF := 0;
  TRISF := 0;

  EEPROM_24C02_Init();                      // performs I2C initialization
  b := 0x00;
  for i := 0x00 to 0x80 do
    begin
      EEPROM_24C02_WrSingle(i,b);
      Inc(b);
      Delay_ms(5); //max vrednost za upis u eeprom
    end;

  for i := 0x00 to 0x80 do
    begin
      LATD := i;
      LATB := EEPROM_24C02_RdSingle(i);
      Delay_ms(100);
    end;
end.
```

Keypad Library

mikroPascal PRO for PIC32 provides a library for working with 4x4 keypad. The library routines can also be used with 4x1, 4x2, or 4x3 keypad. For connections explanation see schematic at the bottom of this page.

External dependencies of Keypad Library

The following variable must be defined in all projects using Keypad Library:	Description:	Example:
`var keypadPort : word; sfr; external;`	Keypad Port.	`var keypadPort : byte at PORTB;`
`var keypadPort_Direction : word; sfr; external;`	Keypad Port.	`var keypadPort_Direction : byte at TRISB;`

Library Routines

- Keypad_Init
- Keypad_Key_Press
- Keypad_Key_Click

Keypad_Init

Prototype	`procedure Keypad_Init();`
Description	Initializes given port for working with keypad.
Parameters	None.
Returns	Nothing.
Requires	Global variable: - `keypadPort` - Keypad port must be defined before using this function.

Example	`// Keypad module connections`
	`var keypadPort : byte at PORTB;`
	`var keypadPort_Direction : byte at TRISB;`
	`// End of keypad module connections`
	`...`
	`Keypad_Init();`

Notes	The Keypad library uses lower byte (bits <7..0>) of `keypadPort`.

Keypad_Key_Press

Prototype	`function Keypad_Key_Press(): word;`
Description	Reads the key from keypad when key gets pressed.
Parameters	None.
Returns	The code of a pressed key (1..16). If no key is pressed, returns 0.
Requires	Port needs to be initialized for working with the Keypad library, see Keypad_Init.

Example	`var kp : word;`
	`...`
	`kp := Keypad_Key_Press();`

Notes	None

Keypad_Key_Click

Prototype	`function Keypad_Key_Click(): word;`
Description	Call to `Keypad_Key_Click` is a blocking call: the function waits until some key is pressed and released. When released, the function returns 1 to 16, depending on the key. If more than one key is pressed simultaneously the function will wait until all pressed keys are released. After that the function will return the code of the first pressed key.
Parameters	None.
Returns	The code of a clicked key (1..16). If no key is clicked, returns 0.
Requires	Port needs to be initialized for working with the Keypad library, see Keypad_Init.
Example	`kp = Keypad_Key_Click();`
Notes	None

Library Example

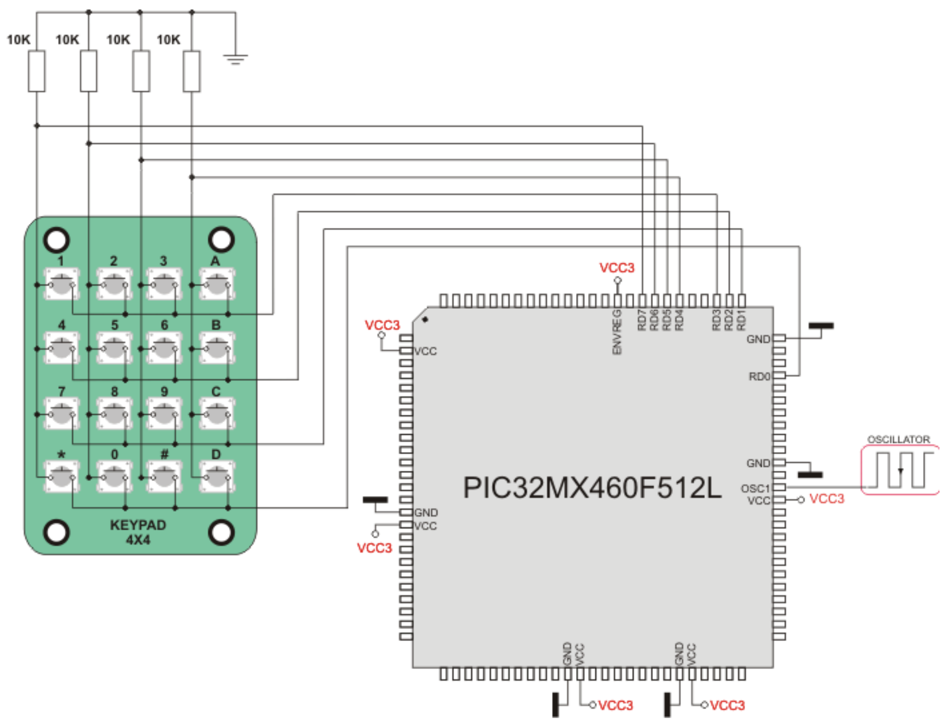
The following code can be used for testing the keypad. It is written for keypad_4x3 or _4x4. The code returned by the keypad functions (1..16) is transformed into ASCII codes [0..9,A..F], and then sent via UART1.

Copy Code To Clipboard

```
program Keypad_Test;
var kp, oldstate : byte;
    txt : array[6] of char;

// Keypad module connections
var keypadPort : dword at PORTD;
var keypadPort_Direction : dword at TRISD;
// End Keypad module connections

  begin
    oldstate := 0;
    AD1PCFG := 0xFFFF;
    CHECON := 0x32;
    UART1_Init(19200);            // Initialize UART module at 9600 bps
    Delay_ms(10);
    Keypad_Init();                         // Initialize Keypad
    UART1_Write_Text('Press any key on your keypad...');
    UART1_Write(10);
    UART1_Write(13);

    while TRUE do
      begin
        kp := 0;                           // Reset key code variable

        // Wait for key to be pressed and released
        while ( kp = 0 ) do
          kp := Keypad_Key_Click();        // Store key code in kp variable

        LATB := kp;
        // Prepare value for output, transform key to it's ASCII value
        case kp of
          //case 10: kp = 42;   // '*'     // Uncomment this block for keypad4x3
          //case 11: kp = 48;   // '0'
          //case 12: kp = 35;   // '#'
          //default: kp += 48;

            1: kp := 49;  // 1             // Uncomment this block for keypad4x4
            2: kp := 50;  // 2
            3: kp := 51;  // 3
            4: kp := 65;  // A
            5: kp := 52;  // 4
            6: kp := 53;  // 5
            7: kp := 54;  // 6
            8: kp := 66;  // B
            9: kp := 55;  // 7
           10: kp := 56;  // 8
           11: kp := 57;  // 9
           12: kp := 67;  // C
```

```
            13: kp := 42;  // *
            14: kp := 48;  // 0
            15: kp := 35;  // #
            16: kp := 68;  // D

        end;

        UART1_Write_Text('Key pressed: ');
        UART1_Write(kp);                         // Send value of pressed button to UART
        UART1_Write(10);
        UART1_Write(13);
      end;
  end.
```

HW Connection



4x4 Keypad connection scheme

Lcd Library

mikroPascal PRO for PIC32 provides a library for communication with Lcds (with HD44780 compliant controllers) through the 4-bit interface. An example of Lcd connections is given on the schematic at the bottom of this page.

For creating a set of custom Lcd characters use Lcd Custom Character Tool.

Library Dependency Tree

Keypad_Key_Click

The following variables must be defined in all projects using Lcd Library:	Description:	Example:
`var LCD_RS : sbit; sfr; external;`	Register Select line.	`var LCD_RS : sbit at LATD0_bit;`
`var LCD_EN : sbit; sfr; external;`	Enable line.	`var LCD_EN : sbit at LATD1_bit;`
`var LCD_D7 : sbit; sfr; external;`	Data 7 line.	`var LCD_D7 : sbit at LATB3_bit;`
`var LCD_D6 : sbit; sfr; external;`	Data 6 line.	`var LCD_D6 : sbit at LATB2_bit;`
`var LCD_D5 : sbit; sfr; external;`	Data 5 line.	`var LCD_D5 : sbit at LATB1_bit;`
`var LCD_D4 : sbit; sfr; external;`	Data 4 line.	`var LCD_D4 : sbit at LATB0_bit;`
`var LCD_RS_Direction : sbit; sfr; external;`	Register Select direction pin.	`var LCD_RS_Direction : sbit at TRISD0_bit;`
`var LCD_EN_Direction : sbit; sfr; external;`	Enable direction pin.	`var LCD_EN_Direction : sbit at TRISD1_bit;`
`var LCD_D7_Direction : sbit; sfr; external;`	Data 7 direction pin.	`var LCD_D7_Direction : sbit at TRISB3_bit;`
`var LCD_D6_Direction : sbit; sfr; external;`	Data 6 direction pin.	`var LCD_D6_Direction : sbit at TRISB2_bit;`
`var LCD_D5_Direction : sbit; sfr; external;`	Data 5 direction pin.	`var LCD_D5_Direction : sbit at TRISB1_bit;`
`var LCD_D4_Direction : sbit; sfr; external;`	Data 4 direction pin.	`var LCD_D4_Direction : sbit at TRISB0_bit;`

Library Routines

- Lcd_Init
- Lcd_Out
- Lcd_Out_Cp
- Lcd_Chr
- Lcd_Chr_Cp
- Lcd_Cmd

Lcd_Init

<table>
<tr><td>Prototype</td><td><code>procedure Lcd_Init();</code></td></tr>
<tr><td>Description</td><td>Initializes Lcd module.</td></tr>
<tr><td>Parameters</td><td>None.</td></tr>
<tr><td>Returns</td><td>Nothing.</td></tr>
<tr><td>Requires</td><td>Global variables:

- <code>LCD_D7</code>: Data bit 7
- <code>LCD_D6</code>: Data bit 6
- <code>LCD_D5</code>: Data bit 5
- <code>LCD_D4</code>: Data bit 4
- <code>LCD_RS</code>: Register Select (data/instruction) signal pin
- <code>LCD_EN</code>: Enable signal pin

- <code>LCD_D7_Direction</code>: Direction of the Data 7 pin
- <code>LCD_D6_Direction</code>: Direction of the Data 6 pin
- <code>LCD_D5_Direction</code>: Direction of the Data 5 pin
- <code>LCD_D4_Direction</code>: Direction of the Data 4 pin
- <code>LCD_RS_Direction</code>: Direction of the Register Select pin
- <code>LCD_EN_Direction</code>: Direction of the Enable signal pin

must be defined before using this function.</td></tr>
<tr><td>Example</td><td><pre>// LCD module connections
var LCD_RS : sbit at LATD0_bit;
var LCD_EN : sbit at LATD1_bit;
var LCD_D4 : sbit at LATB0_bit;
var LCD_D5 : sbit at LATB1_bit;
var LCD_D6 : sbit at LATB2_bit;
var LCD_D7 : sbit at LATB3_bit;

var LCD_RS_Direction : sbit at TRISD0_bit;
var LCD_EN_Direction : sbit at TRISD1_bit;
var LCD_D4_Direction : sbit at TRISB0_bit;
var LCD_D5_Direction : sbit at TRISB1_bit;
var LCD_D6_Direction : sbit at TRISB2_bit;
var LCD_D7_Direction : sbit at TRISB3_bit;
// End LCD module connections

...
Lcd_Init();</pre></td></tr>
<tr><td>Notes</td><td>None</td></tr>
</table>

Lcd_Out

Prototype	`procedure Lcd_Out(row, column: word; var text: string);`
Description	Prints text on Lcd starting from specified position. Both string variables and literals can be passed as a text.
Parameters	- `row`: starting position row number - `column`: starting position column number - `text`: text to be written
Returns	Nothing.
Requires	The Lcd module needs to be initialized. See Lcd_Init routine.
Example	`// Write text "Hello!" on Lcd starting from row 1, column 3:` `Lcd_Out(1, 3, "Hello!");`
Notes	None

Lcd_Out_Cp

Prototype	`procedure Lcd_Out_Cp(var text: string);`
Returns	Nothing.
Description	Prints text on Lcd at current cursor position. Both string variables and literals can be passed as a text.
Parameters	- `text`: text to be written
Requires	The Lcd module needs to be initialized. See Lcd_Init routine.
Example	`// Write text "Here!" at current cursor position:` `Lcd_Out_Cp("Here!");`
Notes	None

Lcd_Chr

Prototype	`procedure Lcd_Chr(row, column: word, out_char: byte);`
Description	Prints character on Lcd at specified position. Both variables and literals can be passed as a character.
Parameters	- `row`: writing position row number - `column`: writing position column number - `out_char`: character to be written
Returns	Nothing.
Requires	The Lcd module needs to be initialized. See Lcd_Init routine.
Example	`// Write character "i" at row 2, column 3:` `Lcd_Chr(2, 3, 'i');`
Notes	None

Lcd_Chr_Cp

Prototype	`procedure Lcd_Chr_Cp(out_char: byte);`
Description	Prints character on Lcd at current cursor position. Both variables and literals can be passed as a character.
Parameters	- `out_char`: character to be written
Returns	Nothing.
Requires	The Lcd module needs to be initialized. See Lcd_Init routine.
Example	`// Write character "e" at current cursor position:` `Lcd_Chr_Cp('e');`
Notes	None

Lcd_Cmd

Prototype	`procedure Lcd_Cmd(out_char: byte);`
Description	Sends command to Lcd.
Parameters	- `out_char`: command to be sent
Returns	Nothing.
Requires	The Lcd module needs to be initialized. See Lcd_Init table.
Example	`// Clear Lcd display:` `Lcd_Cmd(_LCD_CLEAR);`
Notes	Predefined constants can be passed to the function, see Available Lcd Commands.

Available Lcd Commands

Lcd Command	Purpose
`_LCD_FIRST_ROW`	Move cursor to the 1st row
`_LCD_SECOND_ROW`	Move cursor to the 2nd row
`_LCD_THIRD_ROW`	Move cursor to the 3rd row
`_LCD_FOURTH_ROW`	Move cursor to the 4th row
`_LCD_CLEAR`	Clear display
`_LCD_RETURN_HOME`	Return cursor to home position, returns a shifted display to its original position. Display data RAM is unaffected.
`_LCD_CURSOR_OFF`	Turn off cursor
`_LCD_UNDERLINE_ON`	Underline cursor on
`_LCD_BLINK_CURSOR_ON`	Blink cursor on
`_LCD_MOVE_CURSOR_LEFT`	Move cursor left without changing display data RAM
`_LCD_MOVE_CURSOR_RIGHT`	Move cursor right without changing display data RAM
`_LCD_TURN_ON`	Turn Lcd display on
`_LCD_TURN_OFF`	Turn Lcd display off
`_LCD_SHIFT_LEFT`	Shift display left without changing display data RAM
`_LCD_SHIFT_RIGHT`	Shift display right without changing display data RAM

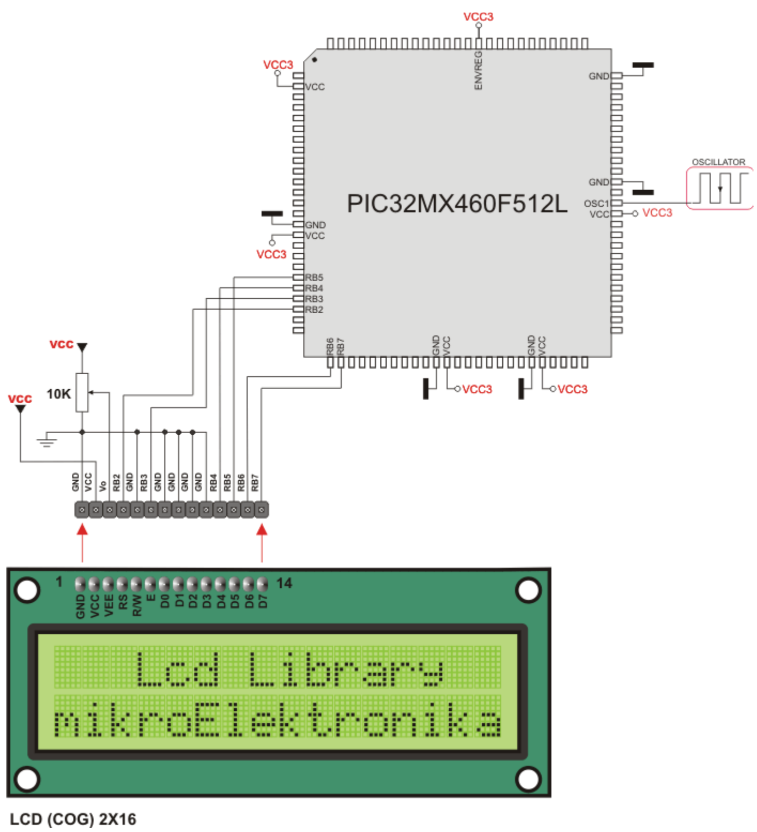
Library Example

The following code demonstrates usage of the Lcd Library routines:

Copy Code To Clipboard

```
program Lcd_COG_2x16;

// LCD module connections
var LCD_RS : sbit  at LATB2_bit;
var LCD_EN : sbit  at LATB3_bit;
var LCD_D4 : sbit  at LATB4_bit;
var LCD_D5 : sbit  at LATB5_bit;
var LCD_D6 : sbit  at LATB6_bit;
var LCD_D7 : sbit  at LATB7_bit;

var LCD_RS_Direction : sbit at TRISB2_bit;
var LCD_EN_Direction : sbit at TRISB3_bit;
var LCD_D4_Direction : sbit at TRISB4_bit;
var LCD_D5_Direction : sbit at TRISB5_bit;
var LCD_D6_Direction : sbit at TRISB6_bit;
var LCD_D7_Direction : sbit at TRISB7_bit;
// End LCD module connections

var txt1 : array[16] of char;
    txt2 : array[10]  of char;
    txt3 : array[8]  of char;
    txt4 : array[7]  of char;
    i    : byte;                      // Loop variable

procedure Move_Delay();               // Function used for text moving
  begin
    Delay_ms(500);                    // You can change the moving speed here
  end;

  begin
    CHECON := 0x32;
    AD1PCFG := 0xFFFF;                // Configure AN pins as digital I/O

    txt1 := 'mikroElektronika';
    txt2 := 'LV32MX v6';
    txt3 := 'Lcd4bit';
    txt4 := 'example';

    Lcd_Init();                       // Initialize LCD
    Lcd_Cmd(_LCD_CLEAR);              // Clear display
    Lcd_Cmd(_LCD_CURSOR_OFF);         // Cursor off
    LCD_Out(1,6,txt3);                // Write text in first row
    LCD_Out(2,6,txt4);                // Write text in second row
    Delay_ms(2000);
    Lcd_Cmd(_LCD_CLEAR);              // Clear display

    LCD_Out(1,1,txt1);                // Write text in first row
    Lcd_Out(2,4,txt2);                // Write text in second row
    Delay_ms(500);
```

```
// Moving text
   for i:=0 to 3 do                        // Move text to the right 4 times
     begin
       Lcd_Cmd(_LCD_SHIFT_RIGHT);
       Move_Delay();
     end;

   while TRUE do                           // Endless loop
     begin
       for i:=0 to 7 do                    // Move text to the left 7 times
         begin
           Lcd_Cmd(_LCD_SHIFT_LEFT);
           Move_Delay();
         end;

       for i:=0 to 7 do                    // Move text to the right 7 times
         begin
           Lcd_Cmd(_LCD_SHIFT_RIGHT);
           Move_Delay();
         end;
     end;
 end.
```



Lcd HW connection

Memory Manager Library

This library provides routines for accessing microcontroller's (internal) Flash memory.

Library Routines

- Heap_Init
- malloc
- free
- LargestFreeMemBlock
- TotalFreeMemSize

Heap_Init

Prototype	`procedure Heap_Init();`
Description	Sets Heap size.
Parameters	None.
Returns	Nothing.
Requires	Nothing.

Example	`const HEAP_SIZE = 3000;   // declare Heap size`
	`Heap_Init();              // set Heap size`

Notes	None.

GetMem

Prototype	`procedure GetMem(var P: ^dword; WantedSize: word);`
Description	Fetches memory from the memory heap.
Parameters	- `WantedSize`: pointer to the fetched memory - `WantedSize`: size in bytes of the dynamic variable to allocate
Returns	Returns a pointer to the fetched memory (of "WantedSize" bytes) in P if success; Otherwise 0 (no free blocks of memory are large enough).
Requires	Nothing.
Example	`GetMem(ptr,20*sizeof(PBuffer));  // ptr will point to a memory block where PBuffer is allocated`
Notes	None.

FreeMem

Prototype	`procedure FreeMem(var P: ^dword; ActualSize: word);`
Description	FreeMem destroys the variable referenced by P and returns its memory to the heap.
Parameters	- `P`: variable of any pointer type previously assigned by the Getmem procedure. - `ActualSize`: specifies the size in bytes of the dynamic variable to dispose of and should be the same as the one used to Getmem.
Returns	Nothing.
Requires	Nothing.
Example	`FreeMem(ptr,20*sizeof(PBuffer));  // ptr will point to a memory block where PBuffer is allocated`
Notes	None.

MM_LargestFreeMemBlock

Prototype	`function MM_LargestFreeMemBlock() : word;`
Description	This function is used to determine largest available free memory block for the Heap.
Parameters	None.
Returns	Returns, after defragmentation of the freelist the size (in bytes) of the largest free block of contiguous memory on the heap.
Requires	Nothing.

Example	`var block : word;`
	
	`begin`
	`  block := MM_LargestFreeMemBlock();`
	`end;`

Notes	None.

MM_TotalFreeMemSize

Prototype	`function MM_TotalFreeMemSize() : word;`
Description	This function is used to determine total free memory size on the heap.
Parameters	None.
Returns	Returns the size (in bytes) of the total free memory on the heap.
Requires	Nothing.

Example	`var total : word;`
	
	`begin`
	`  total := MM_TotalFreeMemSize();`
	`end;`

Notes	None.

Multi Media Card Library

The Multi Media Card (MMC) is a Flash memory card standard. MMC cards are currently available in sizes up to and including 32 GB and are used in cellular phones, digital audio players, digital cameras and PDA's.
mikroPascal PRO for PIC32 provides a library for accessing data on Multi Media Card via SPI communication. This library also supports SD (Secure Digital) and high capacity SDHC (Secure Digital High Capacity) memory cards.

Secure Digital Card

Secure Digital (SD) is a Flash memory card standard, based on the older Multi Media Card (MMC) format.
SD cards are currently available in sizes of up to and including 2 GB, and are used in digital cameras, digital camcorders, handheld computers, media players, mobile phones, GPS receivers, video games and PDAs.

Secure Digital High Capacity Card

SDHC (Secure Digital High Capacity, SD 2.0) is an extension of the SD standard which increases card's storage capacity up to 32 GB by using sector addressing instead of byte addressing in the previous SD standard.
SDHC cards share the same physical and electrical form factor as older (SD 1.x) cards, allowing SDHC-devices to support both newer SDHC cards and older SD-cards. The current standard limits the maximum capacity of an SDHC card to 32 GB.

Important:

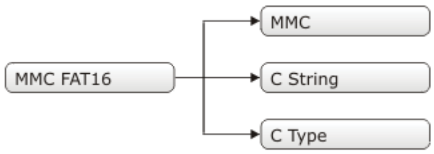
- Routines for file handling can be used only with FAT16 file system.
- Library functions create and read files from the root directory only.
- Library functions populate both FAT1 and FAT2 tables when writing to files, but the file data is being read from the FAT1 table only; i.e. there is no recovery if the FAT1 table gets corrupted.
- If MMC/SD card has Master Boot Record (MBR), the library will work with the first available primary (logical) partition that has non-zero size. If MMC/SD card has Volume Boot Record (i.e. there is only one logical partition and no MBRs), the library works with entire card as a single partition. For more information on MBR, physical and logical drives, primary/secondary partitions and partition tables, please consult other resources, e.g. Wikipedia and similar.
- Before write operation, make sure you don't overwrite boot or FAT sector as it could make your card on PC or digital camera unreadable. Drive mapping tools, such as Winhex, can be of a great assistance.
- Library uses SPI module for communication. The user must initialize the appropriate SPI module before using the MMC Library.
- For MCUs with multiple SPI modules it is possible to initialize all of them and then switch by using the `SPI_Set_Active()` function. See the SPI Library functions.

The SPI module has to be initialized through `SPIx_Init_Advanced` routine with the following parameters:

- SPI Master
- 8bit mode
- secondary prescaler 1
- primary prescaler 64
- Slave Select disabled
- data sampled in the middle of data output time
- clock idle high
- Serial output data changes on transition from active clock state to idle clock state

Tip : Once the MMC/SD card is initialized, SPI module can be reinitialized at higher a speed. See the Mmc_Init and Mmc_Fat_Init routines.

Library Dependency Tree



External dependencies of MMC Library

The following variable must be defined in all projects using MMC library:	Description:	Example:
`var Mmc_Chip_Select : sbit; sfr; external;`	Chip select pin.	`var Mmc_Chip_Select : sbit at LATF0_bit;`
`var Mmc_Chip_Select_Direction : sbit; sfr; external;`	Direction of the chip select pin.	`var Mmc_Chip_Select_Direction : sbit at TRISF0_bit;`

Library Routines

- Mmc_Init
- Mmc_Read_Sector
- Mmc_Write_Sector
- Mmc_Read_Cid
- Mmc_Read_Csd

Routines for file handling:

- Mmc_Fat_Init
- Mmc_Fat_QuickFormat
- Mmc_Fat_Assign
- Mmc_Fat_Reset
- Mmc_Fat_Read
- Mmc_Fat_Rewrite
- Mmc_Fat_Append
- Mmc_Fat_Delete
- Mmc_Fat_Write
- Mmc_Fat_Set_File_Date
- Mmc_Fat_Get_File_Date
- Mmc_Fat_Get_File_Date_Modified
- Mmc_Fat_Get_File_Size
- Mmc_Fat_Get_Swap_File

Mmc_Init

Prototype	`function Mmc_Init(): word;`
Description	Initializes MMC through hardware SPI interface. Mmc_Init needs to be called before using other functions of this library.
Parameters	None.
Returns	- `0` - if MMC/SD card was detected and successfully initialized - `1` - otherwise
Requires	The appropriate hardware SPI module must be previously initialized. Global variables : - `Mmc_Chip_Select`: Chip Select line - `Mmc_Chip_Select_Direction`: Direction of the Chip Select pin must be defined before using this function.
Example	see code below
Notes	None.

```
// MMC module connections
var Mmc_Chip_Select : sbit at LATF0_bit;
var Mmc_Chip_Select_Direction : sbit at TRISF0_bit;
// MMC module connections
...
// Initialize the SPI module
SPI1_Init_Advanced(_SPI_MASTER,  _SPI_8_BIT,  _SPI_PRESCALE_SEC_1,  _SPI_PRESCALE_PRI_64,_SPI_SS_DISABLE,
               _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_HIGH, _SPI_ACTIVE_2_IDLE);
// Loop until MMC is initialized
while (Mmc_Init())
  ;
// Reinitialize the SPI module at higher speed (change primary prescaler).
SPI1_Init_Advanced(_SPI_MASTER,  _SPI_8_BIT,  _SPI_PRESCALE_SEC_1,  _SPI_PRESCALE_PRI_4,_SPI_SS_DISABLE,
               _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_HIGH, _SPI_ACTIVE_2_IDLE);
```

Mmc_Read_Sector

Prototype	`function Mmc_Read_Sector(sector: dword; var dbuff: array[512] of byte): word;`
Description	The function reads one sector (512 bytes) from MMC card.
Parameters	- `sector`: MMC/SD card sector to be read. - `dbuff`: buffer of minimum 512 bytes in length for data storage.
Returns	- `0` - if reading was successful - `1` - if an error occurred
Requires	MMC/SD card must be initialized. See Mmc_Init.

Example	`// read sector 510 of the MMC/SD card`
	`var error : word;`
	`    sectorNo : dword;`
	`    dataBuffer : array[512] of byte;`
	`...`
	`sectorNo := 510;`
	`error := Mmc_Read_Sector(sectorNo, dataBuffer);`

Notes	None.

Mmc_Write_Sector

Prototype	`function Mmc_Write_Sector(sector: dword; var data: array[512] of byte): word;`
Description	The function writes 512 bytes of data to one MMC card sector.
Parameters	- `sector`: MMC/SD card sector to be written to. - `dbuff`: data to be written (buffer of minimum 512 bytes in length).
Returns	- `0` - if writing was successful - `1` - if there was an error in sending write command - `2` - if there was an error in writing (data rejected)
Requires	MMC/SD card must be initialized. See Mmc_Init.

Example	`// write to sector 510 of the MMC/SD card`
	`var error : word;`
	`    sectorNo : dword;`
	`    dataBuffer : array[512] of byte;`
	`...`
	`sectorNo := 510;`
	`error := Mmc_Write_Sector(sectorNo, dataBuffer);`

Notes	None.

Mmc_Read_Cid

Prototype	`function Mmc_Read_Cid(var data_cid: array[16] of byte): word;`
Description	The function reads 16-byte CID register.
Parameters	- `data_cid`: buffer of minimum 16 bytes in length for storing CID register content.
Returns	- `0` - if CID register was read successfully - `1` - if there was an error while reading
Requires	MMC/SD card must be initialized. See Mmc_Init.

Example	`var error : word;`
	`    dataBuffer : array[16] of byte;`
	`...`
	`error := Mmc_Read_Cid(dataBuffer);`

Notes	None.

Mmc_Read_Csd

Prototype	`function Mmc_Read_Csd(var data_for_registers: array[16] of byte): word;`
Description	The function reads 16-byte CSD register.
Parameters	- `data_csd`: buffer of minimum 16 bytes in length for storing CSD register content.
Returns	- `0` - if CSD register was read successfully - `1` - if there was an error while reading
Requires	MMC/SD card must be initialized. See Mmc_Init.

Example	`var error : word;`
	`    dataBuffer : array[16] of byte;`
	`...`
	`error := Mmc_Read_Csd(dataBuffer);`

Notes	None.

Mmc_Fat_Init

Prototype	`function Mmc_Fat_Init(): word;`
Description	Initializes MMC/SD card, reads MMC/SD FAT16 boot sector and extracts necessary data needed by the library.
Parameters	None.
Returns	- `0` - if MMC/SD card was detected and successfully initialized - `1` - if FAT16 boot sector was not found - `255` - if MMC/SD card was not detected
Requires	Global variables : - `Mmc_Chip_Select`: Chip Select line - `Mmc_Chip_Select_Direction`: Direction of the Chip Select pin must be defined before using this function. The appropriate hardware SPI module must be previously initialized. See the SPIx_Init, SPIx_Init_Advanced routines.

Example	`// MMC module connections`
	`var Mmc_Chip_Select : sbit at LATF0_bit;`
	`var Mmc_Chip_Select_Direction : sbit at TRISF0_bit;`
	`// MMC module connections`
	`...`
	`// Initialize the SPI module`
	`SPI1_Init_Advanced(_SPI_MASTER, _SPI_8_BIT, _SPI_PRESCALE_SEC_1, _SPI_PRESCALE_PRI_64,_SPI_SS_DISABLE,`
	`_SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_HIGH, _SPI_ACTIVE_2_IDLE);`
	`// Initialize MMC/SD card and MMC_FAT16 library globals`
	`Mmc_Fat_Init();`
	`// Reinitialize the SPI module at higher speed (change primary prescaler).`
	`SPI1_Init_Advanced(_SPI_MASTER, _SPI_8_BIT, _SPI_PRESCALE_SEC_1, _SPI_PRESCALE_PRI_4,_SPI_SS_DISABLE,`
	`_SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_HIGH, _SPI_ACTIVE_2_IDLE);`

Notes	MMC/SD card has to be formatted to FAT16 file system.

Mmc_Fat_QuickFormat

Prototype	`function Mmc_Fat_QuickFormat(var mmc_fat_label : string[11]) : word;`
Description	Formats to FAT16 and initializes MMC/SD card.
Parameters	- `mmc_fat_label`: volume label (11 characters in length). If less than 11 characters are provided, the label will be padded with spaces. If null string is passed volume will not be labeled
Returns	- `0` - if MMC/SD card was detected, successfully formated and initialized - `1` - if FAT16 format was unseccessful - `255` - if MMC/SD card was not detected
Requires	The appropriate hardware SPI module must be previously initialized.

Example	`// Initialize the SPI module`
	`SPI1_Init_Advanced(_SPI_MASTER, _SPI_8_BIT, _SPI_PRESCALE_SEC_1, _SPI_PRESCALE_PRI_64, _SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_HIGH, _SPI_ACTIVE_2_IDLE);`
	`// Format and initialize MMC/SD card and MMC_FAT16 library globals`
	`Mmc_Fat_QuickFormat('mikroE');`
	`// Reinitialize the SPI module at higher speed (change primary prescaler).`
	`SPI1_Init_Advanced(_SPI_MASTER, _SPI_8_BIT, _SPI_PRESCALE_SEC_1, _SPI_PRESCALE_PRI_4,_SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_HIGH, _SPI_ACTIVE_2_IDLE);`

Notes	This routine can be used instead or in conjunction with Mmc_Fat_Init routine. If MMC/SD card already contains a valid boot sector, it will remain unchanged (except volume label field) and only FAT and ROOT tables will be erased. Also, the new volume label will be set.

Mmc_Fat_Assign

Prototype	`function Mmc_Fat_Assign(var filename: array[12] of char; file_cre_attr: byte): word;`
Description	Assigns file for file operations (read, write, delete...). All subsequent file operations will be applied on an assigned file.
Parameters	- `filename`: name of the file that should be assigned for file operations. File name should be in DOS 8.3 (file_name.extension) format. The file name and extension will be automatically padded with spaces by the library if they have less than length required (i.e. "mikro.tx" -> "mikro .tx "), so the user does no have to take care of that. The file name and extension are case insensitive. The library will convert them to proper case automatically, so the user does not have to take care of that. Also, in order to keep backward compatibility with the first version of this library, file names can be entered as UPPERCASE string of 11 bytes in length with no dot character between file name and extension (i.e. "MIKROELETXT" -> MIKROELE.TXT). In this case last 3 characters of the string are considered to be file extension. - `file_cre_attr`: file creation and attributes flags. Each bit corresponds to the appropriate file attribute: (see table below)
Returns	- `1` - if file already exists or file does not exist but a new file is created. - `0` - if file does not exist and no new file is created.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init.
Example	`// create file with archive attribut if it does not already exist` `Mmc_Fat_Assign('MIKRO007.TXT',0xA0);`
Notes	Long File Names (LFN) are not supported.

Bit	Mask	Description
0	0x01	Read Only
1	0x02	Hidden
2	0x04	System
3	0x08	Volume Label
4	0x10	Subdirectory
5	0x20	Archive
6	0x40	Device (internal use only, never found on disk)
7	0x80	File creation flag. If file does not exist and this flag is set, a new file with specified name will be created.

Mmc_Fat_Reset

Prototype	`procedure Mmc_Fat_Reset(var size: dword);`
Description	Procedure resets the file pointer (moves it to the start of the file) of the assigned file, so that the file can be read.
Parameters	- `size`: buffer to store file size to. After file has been opened for reading, its size is returned through this parameter.
Returns	Nothing.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign.

Example	`var size : dword;`
	`...`
	`Mmc_Fat_Reset(size);`

Notes	None.

Mmc_Fat_Read

Prototype	`procedure Mmc_Fat_Read(var bdata_: byte);`
Description	Reads a byte from the currently assigned file opened for reading. Upon function execution file pointers will be set to the next character in the file.
Parameters	- `bdata`: buffer to store read byte to. Upon this function execution read byte is returned through this parameter.
Returns	Nothing.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign. The file must be opened for reading. See Mmc_Fat_Reset.

Example	`var character : byte;`
	`...`
	`Mmc_Fat_Read(character);`

Notes	None.

Mmc_Fat_Rewrite

Prototype	`procedure Mmc_Fat_Rewrite();`
Description	Opens the currently assigned file for writing. If the file is not empty its content will be erased.
Parameters	None.
Returns	Nothing.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign.
Example	`// open file for writing` `Mmc_Fat_Rewrite();`
Notes	None.

Mmc_Fat_Append

Prototype	`procedure Mmc_Fat_Append();`
Description	Opens the currently assigned file for appending. Upon this function execution file pointers will be positioned after the last byte in the file, so any subsequent file write operation will start from there.
Parameters	None.
Returns	Nothing.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign.
Example	`// open file for appending` `Mmc_Fat_Append();`
Notes	None.

Mmc_Fat_Delete

Prototype	`procedure Mmc_Fat_Delete();`
Description	Deletes currently assigned file from MMC/SD card.
Parameters	None.
Returns	Nothing.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign.
Example	`// delete current file` `Mmc_Fat_Delete();`
Notes	None.

Mmc_Fat_Write

Prototype	`procedure Mmc_Fat_Write(var fdata: array[512] of byte; data_len: word);`
Description	Writes requested number of bytes to the currently assigned file opened for writing.
Parameters	- `fdata`: data to be written. - `data_len`: number of bytes to be written.
Returns	Nothing.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign. The file must be opened for writing. See Mmc_Fat_Rewrite or Mmc_Fat_Append.

Example	`var file_contents : array[42] of byte;`
	`...`
	`Mmc_Fat_Write(file_contents, 42); // write data to the assigned file`

Notes	None.

Mmc_Fat_Set_File_Date

Prototype	`procedure Mmc_Fat_Set_File_Date(year: word; month: byte; day: byte; hours: byte; mins: byte; seconds: byte);`
Description	Sets the date/time stamp. Any subsequent file write operation will write this stamp to the currently assigned file's time/date attributes.
Parameters	- `year`: year attribute. Valid values: 1980-2107 - `month`: month attribute. Valid values: 1-12 - `day`: day attribute. Valid values: 1-31 - `hours`: hours attribute. Valid values: 0-23 - `mins`: minutes attribute. Valid values: 0-59 - `seconds`: seconds attribute. Valid values: 0-59
Returns	Nothing.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign. The file must be opened for writing. See Mmc_Fat_Rewrite or Mmc_Fat_Append.
Example	`// April 1st 2005, 18:07:00` `Mmc_Fat_Set_File_Date(2005, 4, 1, 18, 7, 0);`
Notes	None.

Mmc_Fat_Get_File_Date

Prototype	`procedure Mmc_Fat_Get_File_Date(var year: word; var month: byte; var day: byte; var hours: byte; var mins: byte);`
Description	Reads time/date attributes of the currently assigned file.
Parameters	- `year`: buffer to store year attribute to. Upon function execution year attribute is returned through this parameter. - `month`: buffer to store month attribute to. Upon function execution month attribute is returned through this parameter. - `day`: buffer to store day attribute to. Upon function execution day attribute is returned through this parameter. - `hours`: buffer to store hours attribute to. Upon function execution hours attribute is returned through this parameter. - `mins`: buffer to store minutes attribute to. Upon function execution minutes attribute is returned through this parameter.
Returns	Nothing.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign.

Example	`var year : word;`
	`    month, day, hours, mins : byte;`
	`...`
	`Mmc_Fat_Get_File_Date(year, month, day, hours, mins);`

Notes	None.

Mmc_Fat_Get_File_Date_Modified

Prototype	`procedure Mmc_Fat_Get_File_Date_Modified(var year: word; var month: byte; var day: byte; var hours: byte; var mins: byte);`
Description	Retrieves the last modification date/time for the currently selected file. Seconds are not being retrieved since they are written in 2-sec increments.
Parameters	- `year`: buffer to store year attribute to. Upon function execution year attribute is returned through this parameter. - `month`: buffer to store month attribute to. Upon function execution month attribute is returned through this parameter. - `day`: buffer to store day attribute to. Upon function execution day attribute is returned through this parameter. - `hours`: buffer to store hours attribute to. Upon function execution hours attribute is returned through this parameter. - `mins`: buffer to store minutes attribute to. Upon function execution minutes attribute is returned through this parameter.
Returns	Nothing.
Requires	The file must be assigned, see Mmc_Fat_Assign.

Example	`var year : word;`
	`    month, day, hours, mins : byte;`
	`...`
	`Mmc_Fat_Get_File_Date_Modified(year, month, day, hours, mins);`

Mmc_Fat_Get_File_Size

Prototype	`function Mmc_Fat_Get_File_Size(): dword;`
Description	This function reads size of the currently assigned file in bytes.
Parameters	None.
Returns	This function returns size of active file (in bytes).
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign.

Example	`var my_file_size : dword;`
	`...`
	`my_file_size := Mmc_Fat_Get_File_Size();`

Notes	None.

Mmc_Fat_Get_Swap_File

<table>
<tr><td>Prototype</td><td><code>function Mmc_Fat_Get_Swap_File(sectors_cnt: dword; var filename : string[11]; file_attr : byte) : dword;</code></td></tr>
<tr><td>Description</td><td>This function is used to create a swap file of predefined name and size on the MMC/SD media. If a file with specified name already exists on the media, search for consecutive sectors will ignore sectors occupied by this file. Therefore, it is recommended to erase such file if it already exists before calling this function. If it is not erased and there is still enough space for a new swap file, this function will delete it after allocating new memory space for a new swap file.

The purpose of the swap file is to make reading and writing to MMC/SD media as fast as possible, by using the Mmc_Read_Sector() and Mmc_Write_Sector() functions directly, without potentially damaging the FAT system. The swap file can be considered as a “window” on the media where the user can freely write/read data. It’s main purpose in this library is to be used for fast data acquisition; when the time-critical acquisition has finished, the data can be re-written into a “normal” file, and formatted in the most suitable way.</td></tr>
<tr><td>Parameters</td><td>- <code>sectors_cnt</code>: number of consecutive sectors that user wants the swap file to have.
- <code>filename</code>: name of the file that should be assigned for file operations. File name should be in DOS 8.3 (file_name.extension) format. The file name and extension will be automatically padded with spaces by the library if they have less than length required (i.e. “mikro.tx” -> “mikro .tx “), so the user does no have to take care of that. The file name and extension are case insensitive. The library will convert them to proper case automatically, so the user does not have to take care of that.
Also, in order to keep backward compatibility with the first version of this library, file names can be entered as UPPERCASE string of 11 bytes in length with no dot character between file name and extension (i.e. “MIKROELETXT” -> MIKROELE.TXT). In this case last 3 characters of the string are considered to be file extension.

- <code>file_attr</code>: file creation and attributes flags. Each bit corresponds to the appropriate file attribute:

<table>
<tr><th>Bit</th><th>Mask</th><th>Description</th></tr>
<tr><td>0</td><td>0x01</td><td>Read Only</td></tr>
<tr><td>1</td><td>0x02</td><td>Hidden</td></tr>
<tr><td>2</td><td>0x04</td><td>System</td></tr>
<tr><td>3</td><td>0x08</td><td>Volume Label</td></tr>
<tr><td>4</td><td>0x10</td><td>Subdirectory</td></tr>
<tr><td>5</td><td>0x20</td><td>Archive</td></tr>
<tr><td>6</td><td>0x40</td><td>Device (internal use only, never found on disk)</td></tr>
<tr><td>7</td><td>0x80</td><td>Not used</td></tr>
</table></td></tr>
<tr><td>Returns</td><td>- Number of the start sector for the newly created swap file, if there was enough free space on the MMC/SD card to create file of required size.
- 0 - otherwise.</td></tr>
<tr><td>Requires</td><td>MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init.</td></tr>
</table>

Example	`//-------------- Try to create a swap file with archive atribute, whose size will be at least 1000 sectors.`
	`// If it succeeds, it sends No. of start sector over UART`
	`var size : dword;`
	`...`
	`size := Mmc_Fat_Get_Swap_File(1000, ‘mikroE.txt’, 0x20);`
	`if (size <> 0) then`
	`begin`
	`  UART1_Write(0xAA);`
	`  UART1_Write(Lo(size));`
	`  UART1_Write(Hi(size));`
	`  UART1_Write(Higher(size));`
	`  UART1_Write(Highest(size));`
	`  UART1_Write(0xAA);`
	`end;`

Notes	Long File Names (LFN) are not supported.

Library Example

This project consists of several blocks that demonstrate various aspects of usage of the Mmc_Fat16 library. These are:

- Creation of new file and writing down to it;
- Opening existing file and re-writing it (writing from start-of-file);
- Opening existing file and appending data to it (writing from end-of-file);
- Opening a file and reading data from it (sending it to UART terminal);
- Creating and modifying several files at once;
- Reading file contents;
- Deleting file(s);
- Creating the swap file (see Help for details);

Copy Code To Clipboard

```
program MMC_FAT_Test;

// MMC module connections
var MMC_chip_select : sbit at LATG9_bit; // for writing to output pin always use latch
var MMC_chip_select_direction : sbit at TRISG9_bit;
// eof MMC module connections

const LINE_LEN = 43;

var
  err_txt : string[20];
  file_contents : string[LINE_LEN];

  filename : string[14]; // File names

  character : byte;
  loop, loop2 : byte;
  size : longint;

  buffer : array[512] of byte;
```

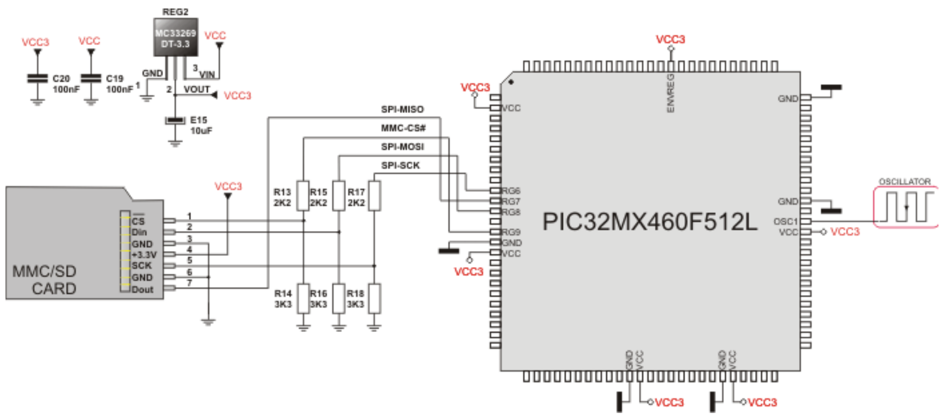
```
// UART write text and new line (carriage return + line feed)
procedure UART_Write_Line( var uart_text : string);
  begin
    UART1_Write_Text(uart_text);
    UART1_Write(13);
    UART1_Write(10);
  end;

//-------------- Creates new file and writes some data to it
procedure M_Create_New_File();
begin
  filename[7] := 'A';                        // Set filename for single-file tests
  Mmc_Fat_Set_File_Date(2011,1,12,11,9,0); // Set file date & time info
  Mmc_Fat_Assign(filename, 0xA0);          // Will not find file and then create file
  Mmc_Fat_Rewrite;                         // To clear file and start with new data
  for loop:=1 to 99 do                     //  We want 5 files on the MMC card
    begin
      UART1_Write('.');
      file_contents[0] := loop div 10 + 48;
      file_contents[1] := loop mod 10 + 48;
      Mmc_Fat_Write(file_contents, LINE_LEN-1);   // write data to the assigned file
    end;
end;

//-------------- Creates many new files and writes data to them
procedure M_Create_Multiple_Files();
begin
  for loop2 := 'B' to 'Z' do
    begin
      UART1_Write(loop2);                       // signal the progress
      filename[7] := loop2;                     // set filename
      Mmc_Fat_Set_File_Date(2011,1,12,11,9,0); // Set file date & time info
      Mmc_Fat_Assign(filename, 0xA0);       // find existing file or create a new one
      Mmc_Fat_Rewrite;                      // To clear file and start with new data
      for loop := 1 to 44 do
        begin
          file_contents[0] := byte(loop div 10 + 48);
          file_contents[1] := byte(loop mod 10 + 48);
          Mmc_Fat_Write(file_contents, LINE_LEN-1); // write data to the assigned file
        end;
    end;
end;

//-------------- Opens an existing file and rewrites it
procedure M_Open_File_Rewrite();
 begin
  filename[7] := 'C';               // Set filename for single-file tests
  Mmc_Fat_Assign(filename, 0);
  Mmc_Fat_Rewrite;
  for loop := 1 to 55 do
   begin
    file_contents[0] := byte(loop div 10 + 48);
    file_contents[1] := byte(loop mod 10 + 48);
    Mmc_Fat_Write(file_contents, 42); // write data to the assigned file
   end;
 end;
```

```
//-------------- Opens an existing file and appends data to it
//               (and alters the date/time stamp)
procedure M_Open_File_Append();
 begin
   filename[7] := 'B';
   Mmc_Fat_Assign(filename, 0);
   Mmc_Fat_Set_File_Date(2009, 1, 23, 17, 22, 0);
   Mmc_Fat_Append();                                   // Prepare file for append
   file_contents := ' for mikroElektronika 2009';      // Prepare file for append
   file_contents[26] := 10;              // LF
   Mmc_Fat_Write(file_contents, 27);     // Write data to assigned file
 end;

//-------------- Opens an existing file, reads data from it and puts it to USART
procedure M_Open_File_Read();
begin
  filename[7] := 'B';
  Mmc_Fat_Assign(filename, 0);
  Mmc_Fat_Reset(size);                   // To read file, procedure returns size of file
  while size > 0 do
    begin
      Mmc_Fat_Read(character);
      UART1_Write(character);            // Write data to UART
      Dec(size);
    end;
end;

//-------------- Deletes a file. If file doesn't exist, it will first be created
//               and then deleted.
procedure M_Delete_File();
begin
  filename[7] := 'F';
  Mmc_Fat_Assign(filename, 0);
  Mmc_Fat_Delete;
end;

//-------------- Tests whether file exists, and if so sends its creation date
//               and file size via USART
procedure M_Test_File_Exist;
var
  fsize: longint;
  year: word;
  month, day, hour, minute: byte;
  outstr: array[12] of char;
begin
  filename[7] := 'B';
  if Mmc_Fat_Assign(filename, 0) <> 0 then
    begin
      //--- file has been found - get its date
      Mmc_Fat_Get_File_Date(year,month,day,hour,minute);
      UART1_Write_Text(' created: ');
      WordToStr(year, outstr);
      UART1_Write_Text(outstr);
      ByteToStr(month, outstr);
      UART1_Write_Text(outstr);
      WordToStr(day, outstr);
```

```
      UART1_Write_Text(outstr);
      WordToStr(hour, outstr);
      UART1_Write_Text(outstr);
      WordToStr(minute, outstr);
      UART1_Write_Text(outstr);

      //--- file has been found - get its modified date
      Mmc_Fat_Get_File_Date_Modified(year, month, day, hour, minute);
      UART1_Write_Text(' modified: ');
      WordToStr(year, outstr);
      UART1_Write_Text(outstr);
      ByteToStr(month, outstr);
      UART1_Write_Text(outstr);
      WordToStr(day, outstr);
      UART1_Write_Text(outstr);
      WordToStr(hour, outstr);
      UART1_Write_Text(outstr);
      WordToStr(minute, outstr);
      UART1_Write_Text(outstr);

      //--- get file size
      fsize := Mmc_Fat_Get_File_Size;
      LongWordToStr(longint(fsize), outstr);
      UART_Write_Line(outstr);
    end
  else
    begin
      //--- file was not found - signal it
      UART1_Write(0x55);
      Delay_ms(1000);
      UART1_Write(0x55);
    end;
end;

//-------------- Tries to create a swap file, whose size will be at least 100
//                 sectors (see Help for details)
procedure M_Create_Swap_File();
  var  i : word;

  begin
    for i:=0 to 511 do
      Buffer[i] := i;

     size := Mmc_Fat_Get_Swap_File(5000, 'mikroE.txt', 0x20);    // see help on this
function for details

    if (size <> 0) then
      begin
        LongIntToStr(size, err_txt);
        UART_Write_Line(err_txt);

       for i:=0 to 4999 do
          begin
            Mmc_Write_Sector(size, Buffer);
            Inc(size);
            UART1_Write('.');
          end;
      end;
  end;
```

```
//-------------- Main. Uncomment the function(s) to test the desired operation(s)
begin
  {$DEFINE COMPLETE_EXAMPLE}       // comment this line to get smaller example
  err_txt := 'FAT16 not found';
  file_contents := 'XX MMC/SD FAT16 library by Anton Rieckert#';
  file_contents[41] := 10;         // newline
  filename := 'MIKRO00xTXT';

  CHECON := 0x32;
  AD1PCFG := 0xFFFF;               // initialize AN pins as digital
  // Initialize UART1 module
  UART1_Init(56000);
  Delay_ms(10);

  UART_Write_Line('MCU-Started');  // MCU present report

  //--- set up SPI for the file read
   SPI2_Init_Advanced(_SPI_MASTER, _SPI_8_BIT, 64, _SPI_SS_DISABLE, _SPI_DATA_SAMPLE_
MIDDLE, _SPI_CLK_IDLE_HIGH, _SPI_ACTIVE_2_IDLE);
  Delay_ms(10);

  // use fat16 quick format instead of init routine if a formatting is needed
  if Mmc_Fat_Init() = 0 then
    begin
      // reinitialize spi at higher speed
      SPI2_Init_Advanced(_SPI_MASTER, _SPI_8_BIT, 8, _SPI_SS_DISABLE, _SPI_DATA_SAMPLE_
MIDDLE, _SPI_CLK_IDLE_HIGH, _SPI_ACTIVE_2_IDLE);
      //--- Test start
      UART_Write_Line('Test Start.');
      //--- Test routines. Uncomment them one-by-one to test certain features
      M_Create_New_File();

      {$IFDEF COMPLETE_EXAMPLE}
      M_Create_New_File();
      M_Create_Multiple_Files();
      M_Open_File_Rewrite();
      M_Open_File_Append();
      M_Open_File_Read();
      M_Delete_File();
      M_Test_File_Exist();
      M_Create_Swap_File();
      {$ENDIF}

      UART_Write_Line('Test End.');
    end
  else
    begin
      UART_Write_Line(err_txt);  // Note: Mmc_Fat_Init tries to initialize a card more
than once.
                               //       If card is not present, initialization may last
longer (depending on clock speed)
    end;
end.
```

HW Connection



Pin diagram of MMC memory card

OneWire Library

The OneWire library provides routines for communication via the Dallas OneWire protocol, for example with DS18x20 digital thermometer. OneWire is a Master/Slave protocol, and all communication cabling required is a single wire. OneWire enabled devices should have open collector drivers (with single pull-up resistor) on the shared data line.

Slave devices on the OneWire bus can even get their power supply from data line. For detailed schematic see device datasheet.

Some basic characteristics of this protocol are:

- single master system,
- low cost,
- low transfer rates (up to 16 kbps),
- fairly long distances (up to 300 meters),
- small data transfer packages.

Each OneWire device also has a unique 64-bit registration number (8-bit device type, 48-bit serial number and 8-bit CRC), so multiple slaves can co-exist on the same bus.

Important :

- Oscillator frequency *Fosc* needs to be at least 4MHz in order to use the routines with Dallas digital thermometers.
- This library implements time-based activities, so interrupts need to be disabled when using OneWire library.

Library Routines

- Ow_Reset
- Ow_Read
- Ow_Write

Ow_Reset

Prototype	`function Ow_Reset(var port: word; pin: word): word;`
Description	Issues OneWire reset signal for DS18x20.
Parameters	- `port`: OneWire bus port - `pin`: OneWire bus pin
Returns	- `0` if the device is present - `1` if the device is not present
Requires	Devices compliant with the Dallas OneWire protocol.
Example	`// Issue Reset signal on One-Wire Bus connected to pin RF6` `Ow_Reset(&PORTF,6);`
Notes	None.

Ow_Read

Prototype	`function Ow_Read(var port : word; pin : word): byte;`
Description	Reads one byte of data via the OneWire bus.
Parameters	- `port`: OneWire bus port - `pin`: OneWire bus pin
Returns	Data read from an external device over the OneWire bus.
Requires	Devices compliant with the Dallas OneWire protocol.

Example	`// Read a byte from the One-Wire Bus connected to pin RF6`
	`var read_data : byte;`
	`...`
	`read_data := Ow_Read(PORTF, 6);`

Notes	None.

Ow_Write

Prototype	`procedure Ow_Write(var port: word; pin, data_ : byte);`
Description	Writes one byte of data via the OneWire bus.
Parameters	- `port`: OneWire bus port - `pin`: OneWire bus pin - `data_`: data to be written
Returns	Nothing.
Requires	Devices compliant with the Dallas OneWire protocol.
Example	`// Send a byte to the One-Wire Bus connected to pin RF6` `Ow_Write(&PORTF, 6, 0xCC);`
Notes	None.

Port Expander Library

mikroPascal PRO for PIC32 provides a library for communication with the Microchip's Port Expander MCP23S17 via SPI interface. Connections of the PIC32 MCU and MCP23S17 is given on the schematic at the bottom of this page.

Important:

- The library uses the SPI module for communication. User must initialize the appropriate SPI module before using the Port Expander Library.
- For MCUs with multiple SPI modules it is possible to initialize all of them and then switch by using the `SPI_Set_Active()` function. See the SPI Library functions.
- Library does not use Port Expander interrupts.

Library Dependency Tree

External dependencies of Port Expander Library

The following variables must be defined in all projects using Port Expander Library:	Description:	Example:
`var SPExpanderRST : sbit; sfr; external;`	Reset line.	`var SPExpanderRST : sbit at LATF0_bit;`
`var SPExpanderCS : sbit; sfr; external;`	Chip Select line.	`var SPExpanderCS : sbit at LATF1_bit;`
`var SPExpanderRST_Direction : sbit; sfr; external;`	Direction of the Reset pin.	`var SPExpanderRST_Direction : sbit at TRISF0_bit;`
`var SPExpanderCS_Direction : sbit; sfr; external;`	Direction of the Chip Select pin.	`var SPExpanderCS_Direction : sbit at TRISF1_bit;`

Library Routines

- Expander_Init
- Expander_Init_Advanced
- Expander_Read_Byte
- Expander_Write_Byte
- Expander_Read_PortA
- Expander_Read_PortB
- Expander_Read_PortAB
- Expander_Write_PortA
- Expander_Write_PortB
- Expander_Write_PortAB
- Expander_Set_DirectionPortA
- Expander_Set_DirectionPortB
- Expander_Set_DirectionPortAB
- Expander_Set_PullUpsPortA
- Expander_Set_PullUpsPortB
- Expander_Set_PullUpsPortAB

Expander_Init

Prototype	`procedure Expander_Init(ModuleAddress : byte);`
Description	Initializes Port Expander using SPI communication. Port Expander module settings: - hardware addressing enabled - automatic address pointer incrementing disabled (byte mode) - BANK_0 register adressing - slew rate enabled
Parameters	- `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page
Returns	Nothing.
Requires	Global variables: - `SPExpanderCS`: Chip Select line - `SPExpanderRST`: Reset line - `SPExpanderCS_Direction`: Direction of the Chip Select pin - `SPExpanderRST_Direction`: Direction of the Reset pin must be defined before using this function. SPI module needs to be initialized. See SPIx_Init and SPIx_Init_Advanced routines.

Example	`// Port Expander module connections`
	`var SPExpanderRST  : sbit at LATF0_bit;`
	`    SPExpanderCS : sbit at LATF1_bit;`
	`    SPExpanderRST_Direction  : sbit at TRISF0_bit;`
	`    SPExpanderCS_Direction : sbit at TRISF1_bit;`
	`// End of Port Expander module connections`
	`...`
	`// If Port Expander Library uses SPI module`
	`SPI1_Init();              // Initialize SPI module used with PortExpander`
	`Expander_Init(0);         // Initialize Port Expander`

Notes	None.

Expander_Init_Advanced

Prototype	`procedure Expander_Init_Advanced(var rstPort : byte; rstPin : byte; haen : byte);`
Description	Initializes Port Expander using SPI communication.
Parameters	- `rstPort`: Port Expander's reset port - `rstPin`: Port Expander's reset pin - `haen`: Port Expander's hardware address
Returns	Nothing.
Requires	- `SPExpanderCS`: Chip Select line - `SPExpanderRST`: Reset line - `SPExpanderCS_Direction`: Direction of the Chip Select pin - `SPExpanderRST_Direction`: Direction of the Reset pin must be defined before using this function. SPI module needs to be initialized. See SPIx_Init and SPIx_Init_Advanced routines.

Example	`// Port Expander module connections`
	`sbit  SPExpanderRST at RF0_bit;`
	`sbit  SPExpanderCS  at RF1_bit;`
	`sbit  SPExpanderRST_Direction at TRISF0_bit;`
	`sbit  SPExpanderCS_Direction  at TRISF1_bit;`
	`// End Port Expander module connections`
	`...`
	
	`// If Port Expander Library uses SPI module`
	`SPI1_Init();        // Initialize SPI1 module used with PortExpander`
	`Expander_Init_Advanced(&PORTB, 0, 0);         // Initialize Port Expander`

Notes	None.

Expander_Read_Byte

Prototype	`function Expander_Read_Byte(ModuleAddress, RegAddress : byte) : byte;`
Description	The function reads byte from Port Expander.
Parameters	- `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `RegAddress`: Port Expander's internal register address
Returns	Byte read.
Requires	Port Expander must be initialized. See Expander_Init.

Example	`// Read a byte from Port Expander's register`
	`var read_data : byte;`
	`...`
	`read_data := Expander_Read_Byte(0,1);`

Notes	None.

Expander_Write_Byte

Prototype	`procedure Expander_Write_Byte(ModuleAddress, RegAddress, Data : byte);`
Description	Routine writes a byte to Port Expander.
Parameters	- `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `RegAddress`: Port Expander's internal register address - `Data`: data to be written
Returns	Byte read.
Requires	Port Expander must be initialized. See Expander_Init.
Example	`// Write a byte to the Port Expander's register` `Expander_Write_Byte(0,1,0xFF);`
Notes	None.

Expander_Read_PortA

Prototype	`function Expander_Read_PortA(ModuleAddress : byte) : byte;`
Description	The function reads byte from Port Expander's PortA.
Parameters	- `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page
Returns	Byte read.
Requires	Port Expander must be initialized. See Expander_Init. Port Expander's PortA should be configured as input. See Expander_Set_DirectionPortA and Expander_Set_DirectionPortAB routines.

Example	`// Read a byte from Port Expander's PORTA`
	`var read_data : byte;`
	`...`
	`Expander_Set_DirectionPortA(0,$FF);   // set expander's porta to be input`
	`...`
	`read_data := Expander_Read_PortA(0);`

Notes	None.

Expander_Read_PortB

Prototype	`function Expander_Read_PortB(ModuleAddress : byte) : byte;`
Description	The function reads byte from Port Expander's PortB.
Parameters	- `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page
Returns	Byte read.
Requires	Port Expander must be initialized. See Expander_Init. Port Expander's PortB should be configured as input. See Expander_Set_DirectionPortB and Expander_Set_DirectionPortAB routines.

Example	`// Read a byte from Port Expander's PORTB`
	`var read_data : byte;`
	`...`
	`Expander_Set_DirectionPortB(0,$FF);        // set expander's portb to be input`
	`...`
	`read_data := Expander_Read_PortB(0);`

Notes	None.

Expander_Read_PortAB

Prototype	`function Expander_Read_PortAB(ModuleAddress : byte) : word;`
Description	The function reads word from Port Expander's ports. PortA readings are in the higher byte of the result. PortB readings are in the lower byte of the result.
Parameters	- `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page
Returns	Word read.
Requires	Port Expander must be initialized. See Expander_Init. Port Expander's PortA and PortB should be configured as inputs. See Expander_Set_DirectionPortA, Expander_Set_DirectionPortB and Expander_Set_DirectionPortAB routines.

Example	`// Read a byte from Port Expander's PORTA and PORTB`
	`var read_data : word;`
	`...`
	`Expander_Set_DirectionPortAB(0,$FFFF);    // set expander's porta and portb to be input`
	`...`
	`read_data := Expander_Read_PortAB(0);`

Notes	None.

Expander_Write_PortA

Prototype	`procedure Expander_Write_PortA(ModuleAddress, Data : byte);`
Description	The function writes byte to Port Expander's PortA.
Parameters	- `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Data`: data to be written
Returns	Nothing.
Requires	Port Expander must be initialized. See Expander_Init. Port Expander's PortA should be configured as output. See Expander_Set_DirectionPortA and Expander_Set_DirectionPortAB routines.

Example	`// Write a byte to Port Expander's PORTA`
	
	`...`
	`Expander_Set_DirectionPortA(0,$00);       // set expander's porta to be output`
	`...`
	`Expander_Write_PortA(0, $AA);`

Notes	None.

Expander_Write_PortB

Prototype	`procedure Expander_Write_PortB(ModuleAddress, Data : byte);`
Description	The function writes byte to Port Expander's PortB.
Parameters	- `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Data`: data to be written
Returns	Nothing.
Requires	Port Expander must be initialized. See Expander_Init. Port Expander's PortB should be configured as output. See Expander_Set_DirectionPortB and Expander_Set_DirectionPortAB routines.

Example	`// Write a byte to Port Expander's PORTB`
	
	`...`
	`Expander_Set_DirectionPortB(0,$00);   // set expander's portb to be output`
	`...`
	`Expander_Write_PortB(0,$55);`

Notes	None.

Expander_Write_PortAB

Prototype	`procedure Expander_Write_PortAB(ModuleAddress : byte; Data : word);`
Description	The function writes word to Port Expander's ports.
Parameters	- `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Data`: data to be written. Data to be written to PortA are passed in Data's higher byte. Data to be written to PortB are passed in `Data`'s lower byte
Returns	Nothing.
Requires	Port Expander must be initialized. See Expander_Init. Port Expander's PortA and PortB should be configured as outputs. See Expander_Set_DirectionPortA, Expander_Set_DirectionPortB and Expander_Set_DirectionPortAB routines.

Example	`// Write a byte to Port Expander's PORTA and PORTB`
	
	`...`
	`Expander_Set_DirectionPortAB(0, $0000);   // set expander's porta and portb to be output`
	`...`
	`Expander_Write_PortAB(0, $AA55);`

Notes	None.

Expander_Set_DirectionPortA

Prototype	`procedure Expander_Set_DirectionPortA(ModuleAddress, Data : byte);`
Description	The function sets Port Expander's PortA direction.
Parameters	- `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Data`: data to be written to the PortA direction register. Each bit corresponds to the appropriate pin of the PortA register. Set bit designates corresponding pin as input. Cleared bit designates corresponding pin as output.
Returns	Nothing.
Requires	Port Expander must be initialized. See Expander_Init.
Example	`// Set Port Expander's PORTA to be output` `Expander_Set_DirectionPortA(0,$00);`
Notes	None.

Expander_Set_DirectionPortB

Prototype	`procedure Expander_Set_DirectionPortB(ModuleAddress, Data : byte);`
Description	The function sets Port Expander's PortB direction.
Parameters	- `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Data`: data to be written to the PortB direction register. Each bit corresponds to the appropriate pin of the PortB register. Set bit designates corresponding pin as input. Cleared bit designates corresponding pin as output.
Returns	Nothing.
Requires	Port Expander must be initialized. See Expander_Init.
Example	`// Set Port Expander's PORTB to be input` `Expander_Set_DirectionPortB(0,$FF);`
Notes	None.

Expander_Set_DirectionPortAB

Prototype	`procedure Expander_Set_DirectionPortAB(ModuleAddress, Direction : word);`
Description	The function sets Port Expander's PortA and PortB direction.
Parameters	- `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Direction`: data to be written to direction registers. Data to be written to the PortA direction register are passed in `Direction`'s higher byte. Data to be written to the PortB direction register are passed in `Direction`'s lower byte. Each bit corresponds to the appropriate pin of the PortA/PortB register. Set bit designates corresponding pin as input. Cleared bit designates corresponding pin as output.
Returns	Nothing.
Requires	Port Expander must be initialized. See Expander_Init.
Example	`// Set Port Expander's PORTA to be output and PORTB to be input` `Expander_Set_DirectionPortAB(0,$00FF);`
Notes	None.

Expander_Set_PullUpsPortA

Prototype	`procedure Expander_Set_PullUpsPortA(ModuleAddress, Data : byte);`
Description	The function sets Port Expander's PortA pull up/down resistors.
Parameters	- `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Data`: data for choosing pull up/down resistors configuration. Each bit corresponds to the appropriate pin of the PortA register. Set bit enables pull-up for corresponding pin.
Returns	Nothing.
Requires	Port Expander must be initialized. See Expander_Init.
Example	`// Set Port Expander's PORTA pull-up resistors` `Expander_Set_PullUpsPortA(0, $FF);`
Notes	None.

Expander_Set_PullUpsPortB

Prototype	`procedure Expander_Set_PullUpsPortB(ModuleAddress, Data : byte);`
Description	The function sets Port Expander's PortB pull up/down resistors.
Parameters	- `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Data`: data for choosing pull up/down resistors configuration. Each bit corresponds to the appropriate pin of the PortB register. Set bit enables pull-up for corresponding pin.
Returns	Nothing.
Requires	Port Expander must be initialized. See Expander_Init.
Example	`// Set Port Expander's PORTB pull-up resistors` `Expander_Set_PullUpsPortB(0, 0xFF);`
Notes	None.

Expander_Set_PullUpsPortAB

Prototype	`procedure Expander_Set_PullUpsPortB(ModuleAddress, Data : byte);`
Description	The function sets Port Expander's PortA and PortB pull up/down resistors.
Parameters	- `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `PullUps`: data for choosing pull up/down resistors configuration. PortA pull up/down resistors configuration is passed in `PullUps`'s higher byte. PortB pull up/down resistors configuration is passed in `PullUps`'s lower byte. Each bit corresponds to the appropriate pin of the PortA/PortB register. Set bit enables pull-up for corresponding pin.
Returns	Nothing.
Requires	Port Expander must be initialized. See Expander_Init.
Example	`// Set Port Expander's PORTB pull-up resistors` `Expander_Set_PullUpsPortB(0, 0xFF);`
Notes	None.

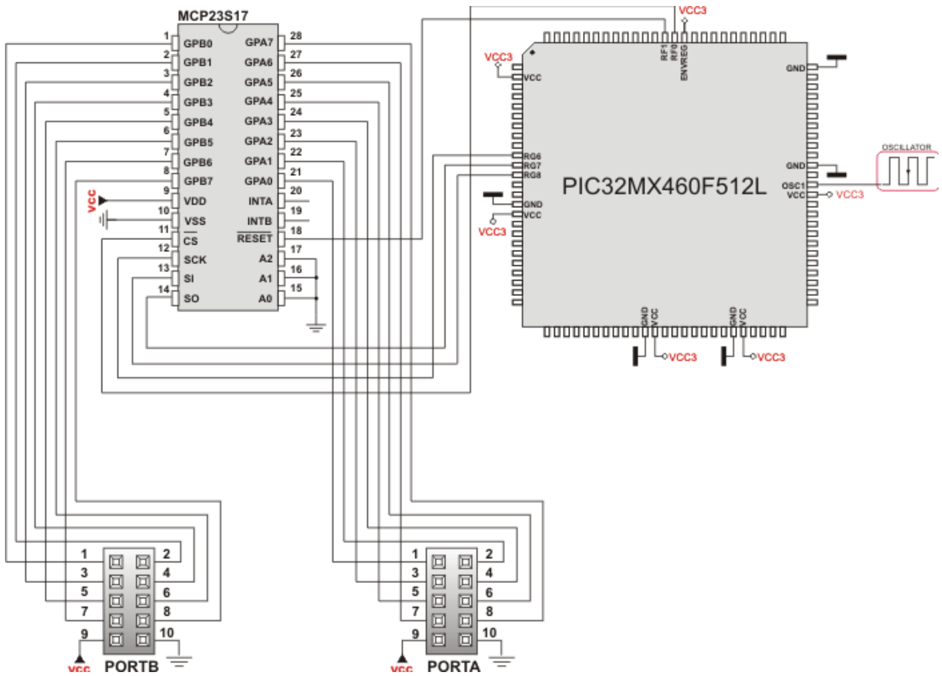
Library Example

The example demonstrates how to communicate with Port Expander MCP23S17. Note that Port Expander pins A2 A1 A0 are connected to GND so Port Expander Hardware Address is 0.

Copy Code To Clipboard

```
program PortExpander;

// Port Expander module connections
var SPExpanderRST : sbit at LATF0_bit;
    SPExpanderCS  : sbit at LATF1_bit;
    SPExpanderRST_Direction : sbit at TRISF0_bit;
    SPExpanderCS_Direction  : sbit at TRISF1_bit;
// End Port Expander module connections

var counter : word;

begin
  counter := 0;
  CHECON := 0x32;
  AD1PCFG := 0xFFFF;                      // Configure AN pins as digital

  TRISB  := 0x00;
  LATB   := 0xFF;

  // If Port Expander Library uses SPI1 module
     SPI2_Init_Advanced(_SPI_MASTER,_SPI_8_BIT,  4,  _SPI_SS_DISABLE,_SPI_DATA_SAMPLE_
MIDDLE,_SPI_CLK_IDLE_LOW,_SPI_ACTIVE_2_IDLE);

  Expander_Init(0);                       // Initialize Port Expander

  Expander_Set_DirectionPortA(0, 0x00);   // Set Expander's PORTA to be output

  Expander_Set_DirectionPortB(0,0xFF);    // Set Expander's PORTB to be input
  Expander_Set_PullUpsPortB(0,0xFF);      // Set pull-ups to all of the Expander's PORTB
pins

  while ( TRUE ) do                       // Endless loop
    begin
      Expander_Write_PortA(0, counter);   // Write i to expander's PORTA
      Inc(counter);
      PORTB := Expander_Read_PortB(0);    // Read expander's PORTB and write it to LEDs
      Delay_ms(75);
    end;

end.
```

HW Connection



Port Expander HW connection

PS/2 Library

The mikroPascal PRO for PIC32 provides a library for communication with the common PS/2 keyboard.

Important:

- The library does not utilize interrupts for data retrieval, and requires the oscillator clock to be at least 6MHz.
- The pins to which a PS/2 keyboard is attached should be connected to the pull-up resistors.
- Although PS/2 is a two-way communication bus, this library does not provide MCU-to-keyboard communication; e.g. pressing the Caps Lock key will not turn on the Caps Lock LED.

External dependencies of PS/2 Library

The following variables must be defined in all projects using PS/2 Library:	Description:	Example:
`var PS2_Data : sbit; sfr; external;`	PS/2 Data line.	`var PS2_Data : sbit at RB0_bit;`
`var PS2_Clock : sbit; sfr; external;`	PS/2 Clock line.	`var PS2_Clock : sbit at RB1_bit;`
`var PS2_Data_Direction : sbit; sfr; external;`	Direction of the PS/2 Data pin.	`var PS2_Data_Direction : sbit at TRISB0_bit;`
`var PS2_Clock_Direction : sbit; sfr; external;`	Direction of the PS/2 Clock pin.	`var PS2_Clock_Direction : sbit at TRISB1_bit;`

Library Routines

- Ps2_Config
- Ps2_Key_Read

Ps2_Config

Prototype	`procedure Ps2_Config();`
Description	Initializes the MCU for work with the PS/2 keyboard.
Parameters	None.
Returns	Nothing.
Requires	Global variables: - `PS2_Data`: Data signal line - `PS2_Clock`: Clock signal line - `PS2_Data_Direction`: Direction of the Data pin - `PS2_Clock_Direction`: Direction of the Clock pin must be defined before using this function.

Example	`// PS2 pinout definition`
	`var PS2_Data : sbit at RB0_bit;`
	`var PS2_Clock : sbit at RB1_bit;`
	`var PS2_Data_Direction : sbit at TRISB0_bit;`
	`var PS2_Clock_Direction : sbit at TRISB1_bit;`
	`// End of PS2 pinout definition// Init PS/2 Keyboard`

Notes	None.

Ps2_Key_Read

Prototype	`function Ps2_Key_Read(var value : byte; var special : byte; var pressed : byte) : word;`
Description	The function retrieves information on key pressed.
Parameters	- `value`: holds the value of the key pressed. For characters, numerals, punctuation marks, and space `value` will store the appropriate ASCII code. Routine “recognizes” the function of Shift and Caps Lock, and behaves appropriately. For special function keys see Special Function Keys Table. - `special`: is a flag for special function keys (F1, Enter, Esc, etc). If key pressed is one of these, `special` will be set to 1, otherwise 0. - `pressed`: is set to 1 if the key is pressed, and 0 if it is released.
Returns	- `1` if reading of a key from the keyboard was successful - `0` if no key was pressed
Requires	PS/2 keyboard needs to be initialized. See Ps2_Config routine.

Example	`var value, special, pressed : word;`
	`...`
	`// Press Enter to continue:`
	`repeat {`
	`  if (Ps2_Key_Read(value, special, pressed)) then`
	`    if ((value = 13) and (special = 1)) then break;`
	`until (0=1);`

Notes	None.

Special Function Keys

Key	Value returned
F1	1
F2	2
F3	3
F4	4
F5	5
F6	6
F7	7
F8	8
F9	9
F10	10
F11	11
F12	12
Enter	13
Page Up	14
Page Down	15
Backspace	16
Insert	17
Delete	18
Windows	19
Ctrl	20
Shift	21
Alt	22
Print Screen	23
Pause	24
Caps Lock	25
End	26
Home	27
Scroll Lock	28
Num Lock	29
Left Arrow	30
Right Arrow	31
Up Arrow	32
Down Arrow	33
Escape	34
Tab	35

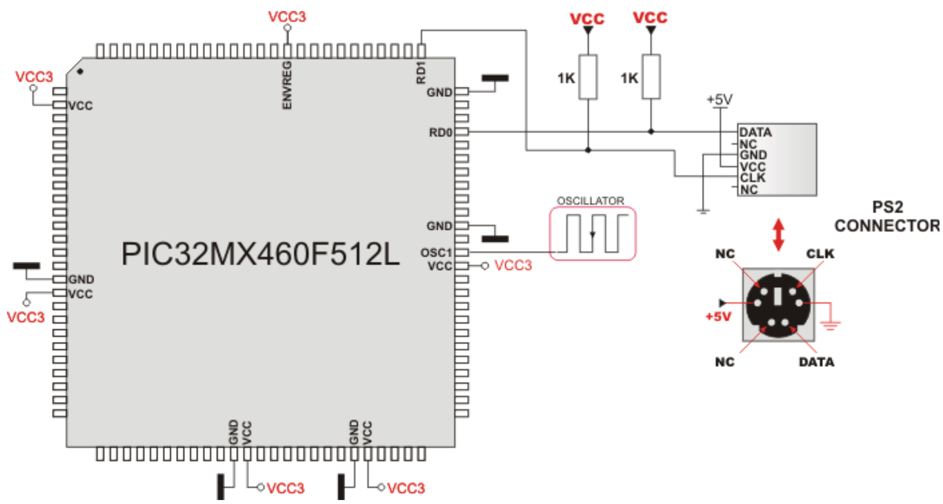
Library Example

This simple example reads values of the pressed keys on the PS/2 keyboard and sends them via UART.

Copy Code To Clipboard

```
program PS2_Example;

var keydata, special, down : byte;

var PS2_Data            : sbit at RD0_bit;
    PS2_Clock           : sbit at RD1_bit;
    PS2_Data_Direction  : sbit at TRISD0_bit;
    PS2_Clock_Direction : sbit at TRISD1_bit;

begin

  CHECON := 0x32;
  AD1PCFG := 0xFFFF;          // Configure AN pins as digital I/O

  UART1_Init(56000);          // Initialize UART module at 19200 bps

  Ps2_Config();               // Init PS/2 Keyboard
  Delay_ms(100);              // Wait for keyboard to finish
  UART1_Write_Text('Ready');  // Ready
  UART1_Write(13);            // Line Feed
  UART1_Write(10);            // Carriage return

    while TRUE do                                        // Endless loop
      begin
        if Ps2_Key_Read(keydata, special, down) then    // If data was read from PS/2
          begin
            if (down <> 0) and (keydata = 16) then      // Backspace
              begin
                UART1_Write(0x08);             // Send Backspace to usart terminal
              end
            else if (down <> 0) and (keydata = 13) then                   // Enter
              begin
                UART1_Write(10);             // Send carriage return to usart terminal
                 UART1_Write(13);              // Uncomment this line if usart terminal
also expects line feed
                                                         // for new line transition
              end
            else if (down <> 0) and (special = 0) and (keydata <> 0) then // Common key
read
              begin
                UART1_Write(keydata);                   // Send key to usart terminal
              end;
          end;
        Delay_ms(1);                                   // Debounce period
      end;
end.
```

HW Connection



Example of PS2 keyboard connection

PWM Library

The CCP module is available with a number of PIC32 MCUs. mikroPascal PRO for PIC32 provides a library which simplifies using of the PWM HW Module.

Important : PWM module uses either Timer2 or Timer3 module.

Library Routines

- PWM_Init
- PWM_Init_Advanced
- PWM_Set_Duty
- PWM_Start
- PWM_Stop

PWM_Init

Prototype	`function PWM_Init(freq_hz : dword; enable_channel_x, timer_prescale, use_timer_x : word) : word;`
Description	Initializes the PWM module with duty ratio 0.
Parameters	- `freq_hz`: PWM frequency in Hz (refer to device datasheet for correct values in respect with Fosc) - `enable_channel_x`: number of PWM channel to be initialized. Refer to MCU's datasheet for available PWM channels - `timer_prescale`: timer prescaler parameter. Valid values: `1, 8, 64,` and `256` - `use_timer_x`: timer to be used with the PWM module. Valid values: `2` (Timer2) and `3` (Timer3)
Returns	- `0xFFFF` - if timer settings are not valid - otherwise returns calculated timer period
Requires	MCU must have the HW PWM Module.

Example	`// Initializes the PWM module at 5KHz, channel 1, no clock prescale, timer2 :`
	`var pwm_period1 : word;`
	`...`
	`pwm_period1 := PWM_Init(5000, 1, 0, 2);`

Notes	Number of available PWM channels depends on MCU. Refer to MCU datasheet for details.

PWM_Init_Advanced

Prototype	`function PWM_Init_Advanced(freq_hz, Fpb_kHz : dword; enable_channel_x, timer_prescale, use_timer_x : word) : word;`
Description	Initializes the PWM module with duty ratio 0.
Parameters	- `freq_hz`: PWM frequency in Hz - `Fpb_kHz`: Peripheral Bus Clock frequency in kHz. - `enable_channel_x`: number of PWM channel to be initialized. Refer to MCU's datasheet for available PWM channels - `timer_prescale`: timer prescaler parameter. Valid values: 1, 8, 64, and 256 use_timer_x: timer to be used with the PWM module. Valid values: 2 (Timer2) and 3 (Timer3)
Returns	- `0xFFFF` - if timer settings are not valid - otherwise returns calculated timer period
Requires	MCU must have the HW PWM Module.
Example	
Notes	Number of available PWM channels depends on MCU. Refer to MCU datasheet for details.

PWM_Set_Duty

Prototype	`procedure PWM_Set_Duty(duty, channel : word);`
Description	The function changes PWM duty ratio.
Parameters	- `duty`: PWM duty ratio. Valid values: `0` to timer period returned by the PWM_Init function. - `channel`: number of PWM channel to change duty to.
Returns	Nothing.
Requires	MCU must have the HW PWM Module. PWM channel must be properly initialized. See PWM_Init routine.

Example	`// Set channel 1 duty ratio to 50%:`
	`var pwm_period1 : word;`
	`...`
	`PWM_Set_Duty(pwm_period1 div 2, 1);`

Notes	Number of available PWM channels depends on MCU. Refer to MCU datasheet for details.

PWM_Start

Prototype	`procedure PWM_Start(enable_channel_x : byte);`
Description	Starts PWM at requested channel.
Parameters	- `enable_channel_x`: number of PWM channel
Returns	Nothing.
Requires	MCU must have the HW PWM Module. PWM channel must be properly configured. See the PWM_Init and PWM_Set_Duty routines.
Example	`// start PWM at channel 1` `PWM_Start(1);`
Notes	Number of available PWM channels depends on MCU. Refer to MCU datasheet for details.

PWM_Stop

Prototype	`procedure PWM_Stop(disable_channel_x : byte);`
Description	Stops PWM at requested channel.
Parameters	- `disable_channel_x`: number of PWM channel
Returns	Nothing.
Requires	MCU must have the HW PWM Module.
Example	`// stop PWM at channel 1` `PWM_Stop(1);`
Notes	Number of available PWM channels depends on MCU. Refer to MCU datasheet for details.

Library Example

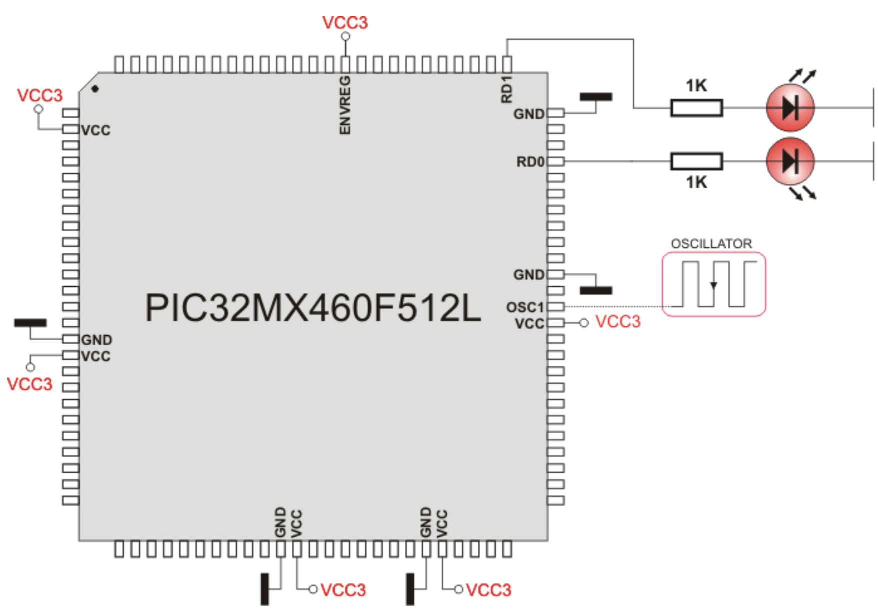
The example changes PWM duty ratio on channels 1 and 2 continuously. If LEDs are connected to channels 1 and 2, a gradual change of emitted light will be noticeable.

Copy Code To Clipboard

```
program Pwm_Demo;
var current_duty, old_duty, current_duty1, old_duty1 : word;
    pwm_period1, pwm_period2 : word;

procedure InitMain();
begin
  CHECON := 0x32;
  AD1PCFG := 0xFFFF;                              // configure pins as digital I/O
  TRISB := 0xFFFF;                                // configure PORTB pins as input
  PORTD := 0;                                     // set PORTD to 0
  TRISD := 0;                                     // designate PORTD pins as output
end;

begin
  InitMain();
  current_duty  := 100;                           // initial value for current_duty
  current_duty1 := 100;                           // initial value for current_duty1

  pwm_period1 := PWM_Init(5000 , 1, 1, 2);
  pwm_period2 := PWM_Init(10000, 2, 1, 3);

  PWM_Start(1);
  PWM_Start(2);

  PWM_Set_Duty(current_duty,  1);                 // Set current duty for PWM1
  PWM_Set_Duty(current_duty1, 2);                 // Set current duty for PWM2

  while (TRUE) do                                 // endless loop
    begin
      if RB0_bit = 1 then                         // button on RB0 pressed
        begin
          Delay_ms(1);
          current_duty := current_duty + 5;       // increment current_duty
          if (current_duty > pwm_period1) then   // if we increase current_duty greater
then possible pwm_period1 value
            current_duty := 0;                    // reset current_duty value to zero

          PWM_Set_Duty(current_duty,  1);         // set newly acquired duty ratio
        end;

      if RB1_bit = 1 then                         // button on RB1 pressed
        begin
          Delay_ms(1);
          current_duty := current_duty - 5;       // decrement current_duty
          if (current_duty > pwm_period1) then   // if we decrease current_duty greater
then possible pwm_period1 value (overflow)
            current_duty := pwm_period1;     // set current_duty to max possible value
```

```
          PWM_Set_Duty(current_duty,  1);         // set newly acquired duty ratio
        end;

      if RB2_bit = 1 then                         // button on RB2 pressed
        begin
          Delay_ms(1);
          current_duty1 := current_duty1 + 5;     // increment current_duty1
         if (current_duty1 > pwm_period2) then  // if we increase current_duty1 greater
then possible pwm_period2 value
            current_duty1 := 0;                   // reset current_duty1 value to zero

          PWM_Set_Duty(current_duty1,  2);        // set newly acquired duty ratio
        end;

      if RB3_bit = 1 then                         // button on RB3 pressed
        begin
          Delay_ms(1);
          current_duty1 := current_duty1 - 5;     // decrement current_duty
         if (current_duty1 > pwm_period2) then  // if we decrease current_duty1 greater
then possible pwm_period1 value (overflow)
            current_duty1 := pwm_period2;       // set current_duty to max possible value

          PWM_Set_Duty(current_duty1,  2);
        end;

      Delay_ms(2);                                // slow down change pace a little
    end;
end.
```

HW Connection



PWM demonstration

RS-485 Library

RS-485 is a multipoint communication which allows multiple devices to be connected to a single bus. mikroPascal PRO for PIC32 provides a set of library routines for comfortable work with RS485 system using Master/Slave architecture. Master and Slave devices interchange packets of information. Each of these packets contains synchronization bytes, CRC byte, address byte and the data. Each Slave has unique address and receives only packets addressed to it. The Slave can never initiate communication.

It is the user's responsibility to ensure that only one device transmits via 485 bus at a time.

The RS-485 routines require the UART module. Pins of UART need to be attached to RS-485 interface transceiver, such as LTC485 or similar (see schematic at the bottom of this page).

Library constants:

- START byte value = `150`
- STOP byte value = `169`
- Address 50 is the broadcast address for all Slaves (packets containing address `50` will be received by all Slaves except the Slaves with addresses `150` and `169`).

Important :

- The library uses the UART module for communication. The user must initialize the appropriate UART module before using the RS-485 Library.
- For MCUs with multiple UART modules it is possible to initialize them and then switch by using the UART_Set_Active routine.

Library Dependency Tree

External dependencies of RS-485 Library

The following variable must be defined in all projects using RS-485 Library:	Description:	Example:
`var RS485_rxtx_pin : sbit; sfr; external;`	Control RS-485 Transmit/Receive operation mode	`var RS485_rxtx_pin : sbit at RF2_bit;`
`var RS485_rxtx_pin_direction : sbit; sfr; external;`	Direction of the RS-485 Transmit/ Receive pin	`var RS485_rxtx_pin_direction : sbit at TRISF2_bit;`

Library Routines

- RS485Master_Init
- RS485Master_Receive
- RS485Master_Send
- RS485Slave_Init
- RS485Slave_Receive
- RS485Slave_Send

RS485Master_Init

Prototype	`procedure RS485Master_Init();`
Description	Initializes MCU as a Master for RS-485 communication.
Parameters	None.
Returns	Nothing.
Requires	Global variables: - `RS485_rxtx_pin` - this pin is connected to RE/DE input of RS-485 transceiver(see schematic at the bottom of this page). RE/DE signal controls RS-485 transceiver operation mode. - `RS485_rxtx_pin_direction` - direction of the RS-485 Transmit/Receive pin. must be defined before using this routine. UART HW module needs to be initialized. See UARTx_Init.

Example	`// RS485 module pinout`
	`var RS485_rxtx_pin : sbit at RF2_bit;`
	`var RS485_rxtx_pin_direction : sbit at TRISF2_bit;`
	`// End of RS485 module pinout`
	`...`
	`UART1_Init(9600);                          // initialize UART1 module`
	`RS485Master_Init();                          // intialize MCU as a Master for RS-485 communication`

Notes	None

RS485Master_Receive

Prototype	`procedure RS485Master_Receive(var data : array[10] of byte);`
Description	Receives messages from Slaves. Messages are multi-byte, so this routine must be called for each byte received.
Parameters	- `data_buffer`: 7 byte buffer for storing received data. Data will be stored in the following manner: - `data_buffer[0..2]`: message content - `data_buffer[3]`: number of message bytes received, 1–3 - `data_buffer[4]`: is set to 255 when message is received - `data_buffer[5]`: is set to 255 if error has occurred - `data_buffer[6]`: address of the Slave which sent the message The routine automatically adjusts `data[4]` and `data[5]` upon every received message. These flags need to be cleared by software.
Returns	Nothing.
Requires	MCU must be initialized as a Master for RS-485 communication. See RS485Master_Init.

Example	`var msg : array[8] of byte;`
	`...`
	`RS485Master_Receive(msg);`

Notes	None

RS485Master_Send

Prototype	`procedure RS485Master_Send(var buffer : array[20] of byte; datalen : byte; slave_address : byte);`
Description	Sends message to Slave(s). Message format can be found at the bottom of this page.
Parameters	- `data_buffer`: data to be sent - `datalen`: number of bytes for transmition. Valid values: 0...3. - `slave_address`: Slave(s) address
Returns	Nothing.
Requires	MCU must be initialized as a Master for RS-485 communication. See RS485Master_Init. It is the user's responsibility to ensure (by protocol) that only one device sends data via 485 bus at a time.

Example	`var msg : array[8] of byte;`
	`...`
	`// send 3 bytes of data to Slave with address 0x12`
	`RS485Master_Send(msg, 3, 0x12);`

Notes	None

RS485Slave_Init

Prototype	`procedure RS485Slave_Init(slave_address : byte);`
Description	Initializes MCU as a Slave for RS-485 communication.
Parameters	- `Slave_address`: Slave address
Returns	Nothing.
Requires	Global variables: - `RS485_rxtx_pin` - this pin is connected to RE/DE input of RS-485 transceiver(see schematic at the bottom of this page). RE/DE signal controls RS-485 transceiver operation mode. Valid values: `1` (for transmitting) and `0` (for receiving - `RS485_rxtx_pin_direction` - direction of the RS-485 Transmit/Receive pin. must be defined before using this routine. UART HW module needs to be initialized. See UARTx_Init.

Example	Initialize MCU as a Slave with address 160:
	
	`// RS485 module pinout`
	`var RS485_rxtx_pin : sbit at RF2_bit;      // transmit/receive control set to PORTC.B2`
	`var RS485_rxtx_pin_direction : sbit at TRISF2_bit;`
	`// End of RS485 module pinout`
	`...`
	`UART1_Init(9600);                    // initialize UART1 module`
	`RS485Slave_Init(160);                 // intialize MCU as a Slave for RS-485 communication with address 160v`

Notes	None

RS485Slave_Receive

Prototype	`procedure RS485Slave_Receive(var data_buffer : array[20] of byte);`
Description	Receives messages from Master. If Slave address and Message address field don't match then the message will be discarded. Messages are multi-byte, so this routine must be called for each byte received.
Parameters	- `data_buffer`: 6 byte buffer for storing received data, in the following manner: - `data_buffer[0..2]`: message content - `data_buffer[3]`: number of message bytes received, 1–3 - `data_buffer[4]`: is set to 255 when message is received - `data_buffer[5]`: is set to 255 if error has occurred The routine automatically adjusts `data[4]` and `data[5]` upon every received message. These flags need to be cleared by software.
Returns	Nothing.
Requires	MCU must be initialized as a Slave for RS-485 communication. See RS485Slave_Init.

Example	`var msg : array[8] of byte;`
	`...`
	`RS485Slave_Read(msg);`

Notes	None

RS485Slave_Send

Prototype	`procedure RS485Slave_Send(var data : array[20] of byte; datalen : byte);`
Description	Sends message to Master. Message format can be found at the bottom of this page.
Parameters	- `data_buffer`: data to be sent - `datalen`: number of bytes for transmition. Valid values: 0...3.
Returns	Nothing.
Requires	MCU must be initialized as a Slave for RS-485 communication. See RS485Slave_Init. It is the user's responsibility to ensure (by protocol) that only one device sends data via 485 bus at a time.

Example	`var msg : array[8] of byte;`
	`...`
	`// send 2 bytes of data to the Master`
	`RS485Slave_Send(msg, 2);`

Notes	None

Library Example

The example demonstrates working with the PIC32 as a Master node in RS-485 communication. Master sends message to Slave with address 160 and waits for a response. After the response is received, the first byte of received data is incremented and sent back to the Slave. The received data is displayed on PORTB while error on receiving (0xAA) and number of consecutive unsuccessful retries are displayed on PORTD. Hardware configurations in this example are made for the LV-32MX v6 board and PIC32MX460F512L.

Copy Code To Clipboard

```
program RS485_Master_Example;

var dat : array[10] of byte;    // buffer for receving/sending messages
    i, j : byte;
    cnt : longint;

var rs485_rxtx_pin  : sbit at RF2_bit;                // set transcieve pin
    rs485_rxtx_pin_direction : sbit at TRISF2_bit;    // set transcieve pin direction

// Interrupt routine
procedure interrupt(); iv IVT_UART_2; ilevel 7; ics ICS_SRS;
begin
  RS485Master_Receive(dat);
  U2RXIF_bit := 0;              // ensure interrupt not pending
end;

  begin
    cnt := 0;

    CHECON  := 0x32;
    AD1PCFG := 0xFFFF;

    PORTB := 0;
    PORTD := 0;
    TRISB := 0;
    TRISD := 0;

    UART2_Init(19200);          // initialize UART2 module
    Delay_ms(100);

    RS485Master_Init();         // initialize MCU as Master

    dat[0] := 0xAA;
    dat[1] := 0xF0;
    dat[2] := 0x0F;
    dat[4] := 0;                // ensure that message received flag is 0
    dat[5] := 0;                // ensure that error flag is 0
    dat[6] := 0;

    U2IP0_bit := 1;             // set interrupt
    U2IP1_bit := 1;             // priority
    U2IP2_bit := 1;             // to 7

    URXISEL1_U2STA_bit := 0; // 0x = Interrupt flag bit is set when a character is received
    U2RXIF_bit := 0;            // ensure interrupt not pending
```

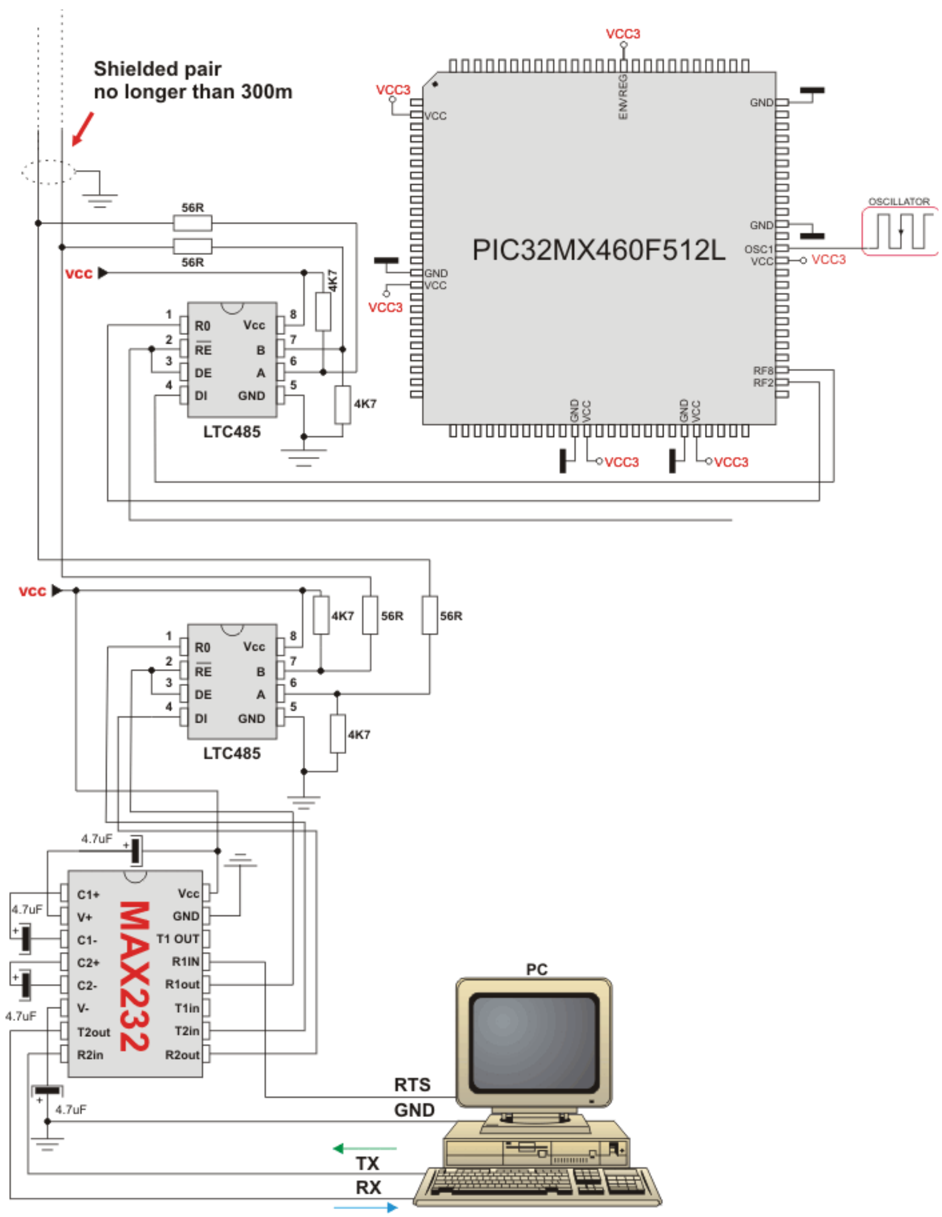
```
    MVEC_bit := 1;          // Interrupt controller configured for multi vectored mode
    asm
      ei R0;                // Enable all interrupts
    end;

    U2RXIE_bit := 1;        // enable intterupt

    RS485Master_Send(dat,1,160);

    while (TRUE) do
      begin                              // upon completed valid message receiving
                                         //   data[4] is set to 255
        Inc(cnt);
        if (dat[5] <> 0) then            // if an error detected, signal it
          PORTD := 0xAA;                 //   by setting portd to 0xAA
        if (dat[4] <> 0) then            // if message received successfully
          begin
            cnt := 0;
            dat[4] := 0;                 // clear message received flag
            j := dat[3];
            for i := 1 to dat[3] do      // show data on PORTB
              PORTB := dat[i-1];
            dat[0] := dat[0]+1;          // send back to master
            Delay_ms(1);
            RS485Master_Send(dat,1,160);
          end;

        if (cnt > 100000) then                // if in 100000 poll-cycles the answer
          begin
            Inc(PORTD);                       //   was not detected, signal
            cnt := 0;                         //   failure of send-message
            RS485Master_Send(dat,1,160);
            if (PORTD > 10) then              // if sending failed 10 times
              begin
                RS485Master_Send(dat,1,50);   //   send message on broadcast address
              end;
          end;
      end;
  end.
```

Copy Code To Clipboard

```
program RS485_Slave_Example;

var dat : array[20] of byte;                // buffer for receving/sending messages
    i, j : byte;

var rs485_rxtx_pin : sbit at RF2_bit;                 // set transcieve pin
    rs485_rxtx_pin_direction : sbit at TRISF2_bit;    // set transcieve pin direction

// Interrupt routine
procedure interrupt(); iv IVT_UART_2; ilevel 7; ics ICS_SRS;
begin
    RS485Slave_Receive(dat);
```

```
    U2RXIF_bit := 0;                      // ensure interrupt not pending
  end;

  begin
    CHECON   := 0x32;
    AD1PCFG := 0xFFFF;

    TRISB := 0;
    TRISD := 0;
    PORTB := 0;
    PORTD := 0;

    UART2_Init(19200);          // initialize UART2 module
    Delay_ms(100);

    RS485Slave_Init(160);       // Intialize MCU as slave, address 160

    dat[0]  := 0xAA;
    dat[1]  := 0xF0;
    dat[2]  := 0x0F;
    dat[4]  := 0;               // ensure that message received flag is 0
    dat[5]  := 0;               // ensure that error flag is 0
    dat[6]  := 0;

    U2IP0_bit := 1;             // set interrupt
    U2IP1_bit := 1;             // priority
    U2IP2_bit := 1;             // to 7

     URXISEL1_U2STA_bit  :=  0;  //  0x  =  Interrupt  flag  bit  is  set  when  a  character  is
received
    U2RXIF_bit := 0;           // ensure interrupt not pending

    MVEC_bit := 1;             // Interrupt controller configured for multi vectored mode
    asm
      ei R0;                   // Enable all interrupts
    end;

    U2RXIE_bit := 1;                      // enable intterupt

    while (TRUE) do
      begin
        if (dat[5] <> 0) then             // if an error detected, signal it by
          begin
            PORTD := 0xAA;                //   setting portd to 0xAA
            dat[5] := 0;
          end;
        if (dat[4] <> 0) then             // upon completed valid message receive
          begin
            dat[4] := 0;                  //   data[4] is set to 0xFF
            j := dat[3];

            for i := 1 to dat[3] do       // show data on PORTB
              PORTB := dat[i-1];

            dat[0] := dat[0]+1;           // increment received dat[0]
            Delay_ms(1);
            RS485Slave_Send(dat,1);       //   and send it back to master
          end;
      end;
  end.
```

HW Connection



Example of interfacing PC to PIC32 MCU via RS485 bus with LTC485 as RS-485 transceiver

Message format and CRC calculations

Q: How is CRC checksum calculated on RS485 master side?

Copy Code To Clipboard

```
const START_BYTE : byte = 0x96;  // 10010110
const STOP_BYTE  : byte = 0xA9;  // 10101001


PACKAGE:
--------
START_BYTE 0x96
ADDRESS
DATALEN
[DATA1]             // if exists
[DATA2]             // if exists
[DATA3]             // if exists
CRC
STOP_BYTE  0xA9


DATALEN bits
------------
bit7 = 1  MASTER SENDS
       0  SLAVE  SENDS
bit6 = 1  ADDRESS WAS XORed with 1, IT WAS EQUAL TO START_BYTE or STOP_BYTE
       0  ADDRESS UNCHANGED
bit5 = 0  FIXED
bit4 = 1  DATA3 (if exists) WAS XORed with 1, IT WAS EQUAL TO START_BYTE or STOP_BYTE
       0  DATA3 (if exists) UNCHANGED
bit3 = 1  DATA2 (if exists) WAS XORed with 1, IT WAS EQUAL TO START_BYTE or STOP_BYTE
       0  DATA2 (if exists) UNCHANGED
bit2 = 1  DATA1 (if exists) WAS XORed with 1, IT WAS EQUAL TO START_BYTE or STOP_BYTE
       0  DATA1 (if exists) UNCHANGED
bit1bit0 = 0 to 3 NUMBER OF DATA BYTES SEND


CRC generation :
----------------
crc_send := datalen  xor address;
crc_send := crc_send xor data[0];    // if exists
crc_send := crc_send xor data[1];    // if exists
crc_send := crc_send xor data[2];    // if exists
crc_send := crc_send not crc_send;
if ((crc_send = START_BYTE) or (crc_send = STOP_BYTE)) then
   crc_send := crc_send + 1;
NOTE: DATALEN<4..0> can not take the START_BYTE<4..0> or STOP_BYTE<4..0> values.
```

Software I²C Library

The mikroPascal PRO for PIC32 provides routines for implementing Software I²C communication. These routines are hardware independent and can be used with any MCU. The Software I²C library enables you to use MCU as Master in I²C communication. Multi-master mode is not supported.

Important:

- This library implements time-based activities, so interrupts need to be disabled when using Software I²C.
- All I²C Library functions are blocking-call functions (they are waiting for I²C clock line to become logical one).
- The pins used for the Software I²C communication should be connected to the pull-up resistors. Turning off the LEDs connected to these pins may also be required.
- Every Software I²C library routine has its own counterpart in Hardware I²C library, except `I2C_Repeated_Start`. `Soft_I2C_Start` is used instead of `I2C_Repeated_Start`.
- Working clock frequency of the Software I²C is 20kHz.

External dependencies of Software I²C Library

The following variable must be defined in all projects using RS-485 Library:	Description:	Example:
`var Soft_I2C_Scl : sbit; sfr; external;`	Soft I²C Clock line.	`var Soft_I2C_Scl : sbit at RF3_bit;`
`var Soft_I2C_Sda : sbit; sfr; external;`	Soft I²C Data line.	`var Soft_I2C_Sda : sbit at RF2_bit;`
`var Soft_I2C_Scl_Direction : sbit; sfr; external;`	Direction of the Soft I²C Clock pin.	`var Soft_I2C_Scl_Direction : sbit at TRISF3_bit;`
`var Soft_I2C_Sda_Direction : sbit; sfr; external;`	Direction of the Soft I²C Data pin.	`var Soft_I2C_Sda_Direction : sbit at TRISF2_bit;`

Library Routines

- Soft_I2C_Init
- Soft_I2C_Start
- Soft_I2C_Read
- Soft_I2C_Write
- Soft_I2C_Stop
- Soft_I2C_Break

Soft_I2C_Init

Prototype	`procedure Soft_I2C_Init();`
Description	Configures the software I²C module.
Parameters	None.
Returns	Nothing.
Requires	Global variables: - `Soft_I2C_Scl`: Soft I²C clock line - `Soft_I2C_Sda`: Soft I²C data line - `Soft_I2C_Scl_Pin_Direction`: Direction of the Soft I²C clock pin - `Soft_I2C_Sda_Pin_Direction`: Direction of the Soft I²C data pin must be defined before using this function.

Example	`// Software I2C connections`
	`var Soft_I2C_Scl           : sbit at RF3_bit;`
	`    Soft_I2C_Sda           : sbit at RF2_bit;`
	`    Soft_I2C_Scl_Direction : sbit at TRISF3_bit;`
	`    Soft_I2C_Sda_Direction : sbit at TRISF2_bit;`
	`// End Software I2C connections`
	`...`
	`Soft_I2C_Init();`

Notes	None

Soft_I2C_Start

Prototype	`procedure Soft_I2C_Start();`
Description	Determines if the I²C bus is free and issues START signal.
Parameters	None.
Returns	Nothing.
Requires	Software I²C must be configured before using this function. See Soft_I2C_Init routine.
Example	`// Issue START signal` `Soft_I2C_Start();`
Notes	None

Soft_I2C_Read

Prototype	`function Soft_I2C_Read(ack : word) : byte;`
Description	Reads one byte from the slave.
Parameters	- `ack`: acknowledge signal parameter. If the `ack==0` not *acknowledge* signal will be sent after reading, otherwise the *acknowledge* signal will be sent.
Returns	One byte from the Slave.
Requires	Soft I²C must be configured before using this function. See Soft_I2C_Init routine. Also, START signal needs to be issued in order to use this function. See Soft_I2C_Start routine.

Example	`var take : byte;`
	`...`
	`// Read data and send the not_acknowledge signal`
	`take := Soft_I2C_Read(0);`

Notes	None

Soft_I2C_Write

Prototype	`function Soft_I2C_Write(data_ : byte) : byte;`
Description	Sends data byte via the I²C bus.
Parameters	- `data_`: data to be sent
Returns	- `0` if there were no errors. - `1` if write collision was detected on the I²C bus.
Requires	Soft I²C must be configured before using this function. See Soft_I2C_Init routine. Also, START signal needs to be issued in order to use this function. See Soft_I2C_Start routine.

Example	`var data_, error : byte;`
	`...`
	`error := Soft_I2C_Write(data_);`
	`error := Soft_I2C_Write($A3);`

Notes	None

Soft_I2C_Stop

Prototype	`procedure Soft_I2C_Stop();`
Description	Issues STOP signal.
Parameters	None.
Returns	Nothing.
Requires	Soft I²C must be configured before using this function. See Soft_I2C_Init routine.
Example	`// Issue STOP signal` `Soft_I2C_Stop();`
Notes	None

Soft_I2C_Break

Prototype	`procedure Soft_I2C_Break();`
Description	All Software I²C Library functions can block the program flow (see note at the top of this page). Calling this routine from interrupt will unblock the program execution. This mechanism is similar to WDT.
Parameters	None.
Returns	Nothing.
Requires	Nothing.
Example	see code below
Notes	Interrupts should be disabled before using Software I²C routines again (see note at the top of this page).

```
var data1, error, counter : byte;

procedure Timer1Int(); org IVT_ADDR_T1INTERRUPT;
begin
  counter := 0;
  if (counter >= 20)
    begin
      Soft_I2C_Break();
      counter := 0;                // reset counter
    end
  else
    Inc(counter);                  // increment counter

  T1IF_bit := 0;                   // Clear Timer1 overflow interrupt flag
end;

begin

  ...

  // try Soft_I2C_Init with blocking prevention mechanism
  IPC0   := IPC0 or 0x1000;        // Interrupt priority level = 1
  T1IE_bit := 1;                   // Enable Timer1 interrupts
  T1CON := 0x8030;                 // Timer1 ON, internal clock FCY, prescaler
1:256
  Soft_I2C_Init();
  T1IE_bit := 0;                   // Disable Timer1 interrupts

end.
```

Library Example

The example demonstrates use of the Software I²C Library. The PIC32 MCU is connected (SCL, SDA pins) to PCF8583 RTC (real-time clock). Program sends date/time to RTC.

Copy Code To Clipboard

```
program RTC_Read;

var seconds, minutes, hours, day, month, year : byte;    // Global date/time variables

// Software I2C connections
var Soft_I2C_Scl           : sbit at RF3_bit;
    Soft_I2C_Sda           : sbit at RF4_bit;
    Soft_I2C_Scl_Direction : sbit at TRISF3_bit;
    Soft_I2C_Sda_Direction : sbit at TRISF4_bit;
// End Software I2C connections

// LCD module connections
var LCD_RS : sbit  at LATB2_bit;
var LCD_EN : sbit  at LATB3_bit;
var LCD_D4 : sbit  at LATB4_bit;
var LCD_D5 : sbit  at LATB5_bit;
var LCD_D6 : sbit  at LATB6_bit;
var LCD_D7 : sbit  at LATB7_bit;

var LCD_RS_Direction : sbit at TRISB2_bit;
var LCD_EN_Direction : sbit at TRISB3_bit;
var LCD_D4_Direction : sbit at TRISB4_bit;
var LCD_D5_Direction : sbit at TRISB5_bit;
var LCD_D6_Direction : sbit at TRISB6_bit;
var LCD_D7_Direction : sbit at TRISB7_bit;
// End LCD module connections

//--------------------- Reads time and date information from RTC (PCF8583)
procedure Read_Time();
  begin
    Soft_I2C_Start();                // Issue start signal
    Soft_I2C_Write(0xA0);            // Address PCF8583, see PCF8583 datasheet
    Soft_I2C_Write(2);               // Start from address 2
    Soft_I2C_Start();                // Issue repeated start signal
    Soft_I2C_Write(0xA1);            // Address PCF8583 for reading R/W=1
    seconds := Soft_I2C_Read(1);     // Read seconds byte
    minutes := Soft_I2C_Read(1);     // Read minutes byte
    hours := Soft_I2C_Read(1);       // Read hours byte
    day := Soft_I2C_Read(1);         // Read year/day byte
    month := Soft_I2C_Read(0);       // Read weekday/month byte}

    Soft_I2C_Stop();                 // Issue stop signal}
  end;
```

```
//-------------------- Formats date and time
procedure Transform_Time();
  begin
    seconds :=  ((seconds and 0xF0) shr 4)*10 + (seconds and 0x0F);// Transform seconds
    minutes :=  ((minutes and 0xF0) shr 4)*10 + (minutes and 0x0F);// Transform months
    hours   :=  ((hours and 0xF0)  shr 4)*10  + (hours and 0x0F);  // Transform hours
    year    :=   (day and 0xC0) shr 6;                             // Transform year
    day     :=  ((day and 0x30) shr 4)*10    + (day and 0x0F);     // Transform day
    month   :=  ((month and 0x10)  shr 4)*10 + (month and 0x0F);   // Transform month
  end;

//-------------------- Output values to LCD
procedure Display_Time();
  begin
     Lcd_Chr(1, 6, (day / 10)    + 48);    // Print tens digit of day variable
     Lcd_Chr(1, 7, (day mod 10)    + 48);  // Print oness digit of day variable
     Lcd_Chr(1, 9, (month / 10) + 48);
     Lcd_Chr(1,10, (month mod 10) + 48);
     Lcd_Chr(1,15,  year         + 49); // Print year vaiable + 1 (start from year 2011)

     Lcd_Chr(2, 6, (hours / 10)    + 48);
     Lcd_Chr(2, 7, (hours mod 10)    + 48);
     Lcd_Chr(2, 9, (minutes / 10) + 48);
     Lcd_Chr(2,10, (minutes mod 10) + 48);
     Lcd_Chr(2,12, (seconds / 10) + 48);
     Lcd_Chr(2,13, (seconds mod 10) + 48);
  end;

//------------------ Performs project-wide init
procedure Init_Main();
  begin
    CHECON := 0x30;
    AD1PCFG := 0xFFFF;          // Initialize AN pins as digital

    Soft_I2C_Init();            // Initialize Soft I2C communication
    Lcd_Init();                 // Initialize LCD
    Lcd_Cmd(_LCD_CLEAR);        // Clear LCD display
    Lcd_Cmd(_LCD_CURSOR_OFF);   // Turn cursor off

    Lcd_Out(1,1,'Date:');       // Prepare and output static text on LCD
    Lcd_Chr(1,8,':');
    Lcd_Chr(1,11,':');
    Lcd_Out(2,1,'Time:');
    Lcd_Chr(2,8,':');
    Lcd_Chr(2,11,':');
    Lcd_Out(1,12,'201');
  end;

//----------------- Main procedure
  begin

   Delay_ms(1000);

   Init_Main();                 // Perform initialization

    while TRUE do               // Endless loop
      begin
        Read_Time();            // Read time from RTC(PCF8583)
        Transform_Time();       // Format date and time
        Display_Time();         // Prepare and display on LCD
      end;
  end.
```

Software SPI Library

The mikroPascal PRO for PIC32 provides routines for implementing Software SPI communication. These routines are hardware independent and can be used with any MCU. The Software SPI Library provides easy communication with other devices via SPI: A/D converters, D/A converters, MAX7219, LTC1290, etc.

Library configuration:

- SPI to Master mode
- Clock value = 20 kHz.
- Data sampled at the middle of interval.
- Clock idle state low.
- Data sampled at the middle of interval.
- Data transmitted at low to high edge.

The library configures SPI to the master mode, clock = 20kHz, data sampled at the middle of interval, clock idle state low and data transmitted at low to high edge.

Important : The Software SPI library implements time-based activities, so interrupts need to be disabled when using it.

External dependencies of Software SPI Library

The following variables must be defined in all projects using Software SPI Library:	Description:	Example:
`var SoftSpi_SDI : sbit; sfr; external;`	Data In line.	`var SoftSpi_SDI : sbit at RF2_bit;`
`var SoftSpi_SDO : sbit; sfr; external;`	Data Out line.	`var SoftSpi_SDO : sbit at LATF3_bit;`
`var SoftSpi_CLK : sbit; sfr; external;`	Clock line.	`var SoftSpi_CLK : sbit at LATF6_bit;`
`var SoftSpi_SDI_Direction : sbit; sfr; external;`	Direction of the Data In pin.	`var SoftSpi_SDI_Direction : sbit at TRISF2_bit;`
`var SoftSpi_SDO_Direction : sbit; sfr; external;`	Direction of the Data Out pin	`var SoftSpi_SDO_Direction : sbit at TRISF3_bit;`
`var SoftSpi_CLK_Direction : sbit; sfr; external;`	Direction of the Clock pin.	`var SoftSpi_CLK_Direction : sbit at TRISF6_bit;`

Library Routines

- Soft_SPI_Init
- Soft_SPI_Read
- Soft_SPI_Write

Soft_SPI_Init

Prototype	`procedure Soft_SPI_Init();`
Description	Routine initializes the software SPI module.
Parameters	None.
Returns	Nothing.
Requires	Global variables: - `SoftSpi_SDI`: Data in line - `SoftSpi_SDO`: Data out line - `SoftSpi_CLK`: Data clock line - `SoftSpi_SDI_Direction`: Direction of the Data in pin - `SoftSpi_SDO_Direction`: Direction of the Data out pin - `SoftSpi_CLK_Direction`: Direction of the Data clock pin must be defined before using this function.

Example	`// Software SPI module connections`
	`var SoftSpi_SDI : sbit at RF2_bit;`
	`var SoftSpi_SDO : sbit at LATF3_bit;`
	`var SoftSpi_CLK : sbit at LATF6_bit;`
	
	`var SoftSpi_SDI_Direction : sbit at TRISF2_bit;`
	`var SoftSpi_SDO_Direction : sbit at TRISF3_bit;`
	`var SoftSpi_CLK_Direction : sbit at TRISF6_bit;`
	`// End Software SPI module connections`
	`...`
	`Soft_SPI_Init(); // Init Soft_SPI`

Notes	None

Soft_SPI_Read

Prototype	`function Soft_SPI_Read(data_ : byte) : byte;`
Description	This routine performs 3 operations simultaneously. It provides clock for the Software SPI bus, reads a byte and sends a byte.
Parameters	- `sdata`: data to be sent.
Returns	Byte received via the SPI bus.
Requires	Soft SPI must be initialized before using this function. See Soft_SPI_Init routine.

Example	`var data_read, data_send : byte;`
	`...`
	`// Read a byte and assign it to data_read variable`
	`// (data_send byte will be sent via SPI during the Read operation)`
	`data_read := Soft_SPI_Read(data_send);`

Notes	None

Soft_SPI_Write

Prototype	`procedure Soft_SPI_Write(sdata : byte);`
Description	This routine sends one byte via the Software SPI bus.
Parameters	- `sdata`: data to be sent.
Returns	Nothing.
Requires	Soft SPI must be initialized before using this function. See Soft_SPI_Init.
Example	`// Write a byte to the Soft SPI bus` `Soft_SPI_Write(0xAA);`
Notes	None

Library Example

This code demonstrates using library routines for Soft_SPI communication. Also, this example demonstrates working with max7219. Eight 7 segment displays are connected to MAX7219. MAX7219 is connected to SDO, SDI, SCK pins are connected accordingly.

Copy Code To Clipboard

```
// DAC module connections
var Chip_Select : sbit at LATD0_bit;
    SoftSpi_CLK : sbit at LATD6_bit;
    SoftSpi_SDI : sbit at RD2_bit;
    SoftSpi_SDO : sbit at LATD3_bit;

var Chip_Select_Direction : sbit at TRISD0_bit;
    SoftSpi_CLK_Direction : sbit at TRISD6_bit;
    SoftSpi_SDI_Direction : sbit at TRISD2_bit;
    SoftSpi_SDO_Direction : sbit at TRISD3_bit;
// End DAC module connections
```

```
var value : word;

procedure InitMain();
  begin
    TRISB0_bit := 1;                          // Set RB0 pin as input
    TRISB1_bit := 1;                          // Set RB1 pin as input
    Chip_Select := 1;                         // Deselect DAC
    Chip_Select_Direction := 0;               // Set CS# pin as Output
    Soft_Spi_Init();                          // Initialize Soft_SPI
  end;

// DAC increments (0..4095) --> output voltage (0..Vref)
procedure DAC_Output( valueDAC : word);
var temp : byte; volatile;
  begin
    Chip_Select := 0;                          // Select DAC chip

    // Send High Byte
    temp := word(valueDAC shr 8) and 0x0F;     // Store valueDAC[11..8] to temp[3..0]
    temp := temp or 0x30;                    // Define DAC setting, see MCP4921 datasheet
    Soft_SPI_Write(temp);                      // Send high byte via Soft SPI

    // Send Low Byte
    temp := valueDAC;                          // Store valueDAC[7..0] to temp[7..0]
    Soft_SPI_Write(temp);                      // Send low byte via Soft SPI

    Chip_Select := 1;                          // Deselect DAC chip
  end;

  begin

    CHECON := 0x32;
    AD1PCFG := 0xFFFF;                         // Configure AN pins as digital

    InitMain();                                // Perform main initialization

    value := 2048;                             // When program starts, DAC gives
                                               //   the output in the mid-range

    while (TRUE) do                            // Endless loop
      begin

        if ((RB0_bit) and (value < 4095)) then   // If RB0 button is pressed
          Inc(value)                             //   increment value
        else
          begin
            if ((RB1_bit) and (value > 0)) then  // If RB1 button is pressed
              Dec(value);                        //   decrement value
          end;

        DAC_Output(value);                       // Send value to DAC chip
        Delay_ms(1);                             // Slow down key repeat pace
      end;
  end.
```

Software UART Library

The mikroPascal PRO for PIC32 provides routines for implementing Software UART communication. These routines are hardware independent and can be used with any MCU.
The Software UART Library provides easy communication with other devices via the RS232 protocol.

Important: The Software UART library implements time-based activities, so interrupts need to be disabled when using it.

Library Routines

- Soft_UART_Init
- Soft_UART_Read
- Soft_UART_Write
- Soft_UART_Break

Soft_UART_Init

Prototype	`function Soft_UART_Init(var port: word; rx, tx: word; baud_rate : dword; inverted : word) : byte;`
Description	Configures and initializes the software UART module. Software UART routines use Delay_Cyc routine. If requested baud rate is too low then calculated parameter for calling `Delay_Cyc` exceeds `Delay_Cyc` argument range. If requested baud rate is too high then rounding error of `Delay_Cyc` argument corrupts Software UART timings.
Parameters	- `port`: software UART port address - `rx`: receiver pin - `tx`: transmiter pin - `baud_rate`: requested baudrate. Maximum baud rate depends on the MCU's clock and working conditions - `inverted`: if set to non-zero value, indicates inverted logic on output
Returns	- `2` - error, requested baud rate is too low - `1` - error, requested baud rate is too high - `0` - successful initialization
Requires	Nothing.

Example	This will initialize software UART and establish the communication at 9600 bps:
	
	`// Initialize Software UART communication on pins RF2(Rx), RF3(Tx), at 14400 bps`
	`Soft_UART_Init(PORTF, 2, 3, 14400, 0);`

Notes	The Software UART library implements time-based activities, so interrupts need to be disabled when using it.

Soft_UART_Read

Prototype	`function Soft_UART_Read(var error : byte) : byte;`
Description	The function receives a byte via software UART. This is a blocking function call (waits for start bit). Programmer can unblock it by calling Soft_UART_ Break routine.
Parameters	- `error`: Error flag. Error code is returned through this variable. Values : - `0` - no error - `1` - stop bit error - `255` - user abort, Soft_UART_Break called
Returns	Byte received via UART.
Requires	Software UART must be initialized before using this function. See the Soft_UART_Init routine.

Example	`var data_ : byte;`
	`    error : word;`
	`...`
	`// wait until data is received`
	`repeat`
	`  data_ := Soft_UART_Read(error);`
	`until (error = 0);`
	
	`// Now we can work with data:`
	`if (data_) then begin  ...  end`

Notes	The Software UART library implements time-based activities, so interrupts need to be disabled when using it.

Soft_UART_Write

Prototype	`procedure Soft_UART_Write(udata : byte);`
Description	This routine sends one byte via the Software UART bus.
Parameters	- `udata`: data to be sent.
Returns	Nothing.
Requires	Software UART must be initialized before using this function. See the Soft_UART_Init routine. Be aware that during transmission, software UART is incapable of receiving data – data transfer protocol must be set in such a way to prevent loss of information.

Example	`var some_byte : byte;`
	`...`
	`some_byte := $0A;`
	`// Write a byte via Soft UART`
	`Soft_UART_Write(some_byte);`

Notes	The Software UART library implements time-based activities, so interrupts need to be disabled when using it.

Soft_UART_Break

Prototype	`procedure Soft_UART_Break();`
Description	Soft_UART_Read is blocking routine and it can block the program flow. Calling Soft_UART_Break routine from the interrupt will unblock the program execution. This mechanism is similar to WDT.
Parameters	None.
Returns	Nothing.
Requires	Nothing.
Example	see code below
Notes	The Software UART library implements time-based activities, so interrupts need to be disabled when using it.

```
var data1, error, counter : byte;

procedure Timer1Int(); org IVT_ADDR_T1INTERRUPT;
begin
  counter := 0;
  if (counter >= 20) then
    begin
      Soft_UART_Break();
      counter := 0;              // reset counter
    end
  else
    Inc(counter);                  // increment counter

  T1IF_bit := 0;                 // Clear Timer1 overflow interrupt flag
end;

begin

  ...

  if (Soft_UART_Init(PORTF, 2, 3, 14400, 0) = 0)
    Soft_UART_Write(0x55);

  ...

  // try Soft_UART_Read with blocking prevention mechanism
  IPC0   := IPC0 or 0x1000;       // Interrupt priority level = 1
  T1IE_bit := 1;                 // Enable Timer1 interrupts
  T1CON := 0x8030;              // Timer1 ON, internal clock FCY, prescaler 
1:256
  data1 := Soft_UART_Read(&error);
  T1IE_bit := 0;                 // Disable Timer1 interrupts
end.
```

Library Example

This example demonstrates simple data exchange via software UART. If MCU is connected to the PC, you can test the example from the mikroPascal PRO for PIC32 USART communication terminal, launch it from the drop-down menu **Tools › USART Terminal** or simply click the USART Terminal Icon .

Copy Code To Clipboard

```
program Soft_UART;

var error : byte;
    counter, byte_read : byte;             // Auxiliary variables

begin
  CHECON := 0x32;
  AD1PCFG := 0xFFFF;                       // Configure AN pins as digital I/O

  TRISB := 0x00;                          // Set PORTB as output (error signalization)
  PORTB := 0;                              // No error

  error := Soft_UART_Init(PORTF, 2, 8, 56000, 0); // Initialize Soft UART at 56000 bps

  if (error > 0) then
    begin
      PORTB := error;                      // Signalize Init error
      while (TRUE) do nop;                 // Stop program
    end;
  Delay_ms(100);

  for counter := ‘z’ downto ‘A’ do         // Send bytes from ‘z’ downto ‘A’
    begin
      Soft_UART_Write(counter);
      Delay_ms(100);
    end;

  while TRUE do                            // Endless loop
    begin
      byte_read := Soft_UART_Read(error);  // Read byte, then test error flag
      if (error <> 0) then                 // If error was detected
        PORTB := error                     //   signal it on PORTB
      else
        Soft_UART_Write(byte_read);      // If error was not detected, return byte read
    end;
end.
```

Sound Library

The mikroPascal PRO for PIC32 provides a Sound Library to supply users with routines necessary for sound signalization in their applications. Sound generation needs additional hardware, such as piezo-speaker (example of piezo-speaker interface is given on the schematic at the bottom of this page).

Library Routines

- Sound_Init
- Sound_Play

Sound_Init

Prototype	`procedure Sound_Init(var snd_port, snd_pin: word);`
Description	Configures the appropriate MCU pin for sound generation.
Parameters	- `snd_port`: sound output port address - `snd_pin`: sound output pin
Returns	Nothing.
Requires	Nothing.
Example	`// Initialize the pin RD3 for playing sound` `Sound_Init(PORTD, 3);`
Notes	None.

Sound_Play

Prototype	`procedure Sound_Play(freq_in_hz, duration_ms: word);`
Description	Generates the square wave signal on the appropriate pin.
Parameters	- `freq_in_hz`: signal frequency in Hertz (Hz) - `duration_ms`: signal duration in miliseconds (ms)
Returns	Nothing.
Requires	In order to hear the sound, you need a piezo speaker (or other hardware) on designated port. Also, you must call Sound_Init to prepare hardware for output before using this function.
Example	`// Play sound of 1KHz in duration of 100ms` `Sound_Play(1000, 100);`
Notes	None.

Library Example

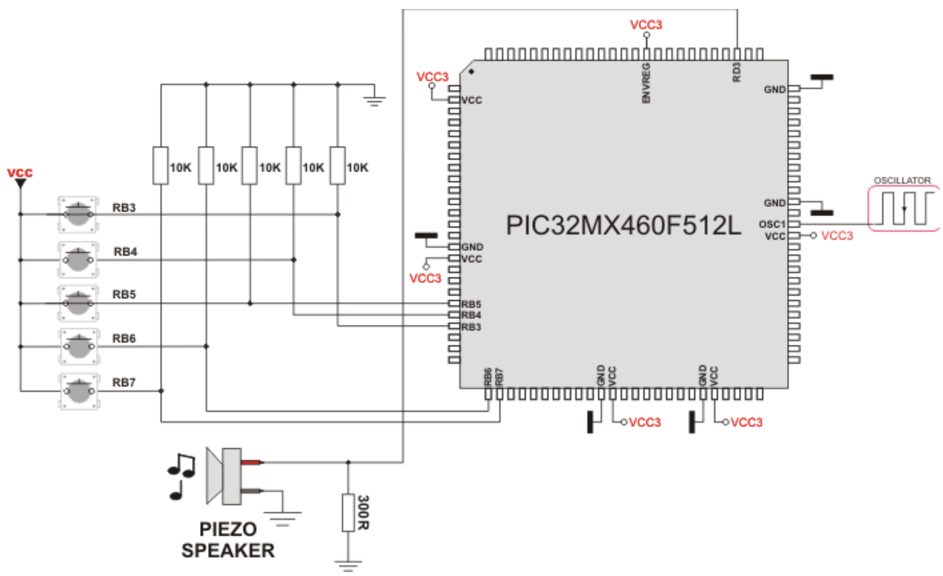
The example is a simple demonstration of how to use the Sound Library for playing tones on a piezo speaker.

Copy Code To Clipboard

```
program Sound;

procedure Tone1();
  begin
    Sound_Play(659, 250);            // Frequency = 659Hz, duration = 250ms
  end;

procedure Tone2();
  begin
    Sound_Play(698, 250);            // Frequency = 698Hz, duration = 250ms
  end;

procedure Tone3();
  begin
    Sound_Play(784, 250);            // Frequency = 784Hz, duration = 250ms
  end;

procedure Melody();                  // Plays the melody "Yellow house"
  begin
    Tone1(); Tone2(); Tone3(); Tone3();
    Tone1(); Tone2(); Tone3(); Tone3();
    Tone1(); Tone2(); Tone3();
    Tone1(); Tone2(); Tone3(); Tone3();
    Tone1(); Tone2(); Tone3();
    Tone3(); Tone3(); Tone2(); Tone2(); Tone1();
  end;

procedure ToneA();                   // Tones used in Melody2 function
  begin
    Sound_Play( 880, 50);
  end;

procedure ToneC();
  begin
    Sound_Play(1046, 50);
  end;

procedure ToneE();
  begin
    Sound_Play(1318, 50);
  end;

procedure Melody2();                 // Plays Melody2
var counter : byte;
  begin
    for counter := 9 downto 1 do
```

```
      begin
        ToneA();
        ToneC();
        ToneE();
      end;
  end;

begin
  CHECON := 0x32;
  AD1PCFG := 0xFFFF;                      // Configure AN pins as digital I/O

  TRISB  := 0xF8;                         // Configure RB7..RB3 as input

  Sound_Init(PORTD, 3);
  Sound_Play(880, 1000);

  while TRUE do                           // endless loop
    begin
      if (Button(PORTB,7,1,1)) then       // If PORTB.7 is pressed play Tone1
        begin
          Tone1();
          while (RB7_bit <> 0) do nop;    // Wait for button to be released
        end;

      if (Button(PORTB,6,1,1)) then       // If PORTB.6 is pressed play Tone1
        begin
          Tone2();
          while (RB6_bit <> 0) do nop;    // Wait for button to be released
        end;

      if (Button(PORTB,5,1,1)) then       // If PORTB.5 is pressed play Tone1
        begin
          Tone3();
          while (RB5_bit <> 0) do nop;    // Wait for button to be released
        end;

      if (Button(PORTB,4,1,1)) then       // If PORTB.4 is pressed play Tone1
        begin
          Melody2();
          while (RB4_bit <> 0) do nop;    // Wait for button to be released
        end;

      if (Button(PORTB,3,1,1)) then       // If PORTB.3 is pressed play Tone1
          begin
            Melody();
            while (RB3_bit <> 0) do nop;  // Wait for button to be released
          end;
    end;
end.
```

HW Connection



Example of Sound Library

SPI Library

The SPI module is available with all PIC32 MCUs. mikroPascal PRO for PIC32 provides a library for initializing the Slave mode and initializing and comfortable work with the Master mode. The PIC32 can easily communicate with other devices via SPI: A/D converters, D/A converters, MAX7219, LTC1290, etc.

Important:

- SPI library routines require you to specify the module you want to use. To select the desired SPI module, simply change the letter **x** in the routine prototype for a number from **1** to **3**.
- Number of SPI modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.
- Switching between the SPI modules in the SPI library is done by the SPI_Set_Active function (both SPI modules have to be previously initialized).

Library Routines

- SPIx_Init
- SPIx_Init_Advanced
- SPIx_Read
- SPIx_Write
- SPI_Set_Active

SPIx_Init

Prototype	`procedure SPIx_Init();`
Description	Configures and initializes the SPI module with default settings. Default settings: - Master mode. - 8-bit data mode. - Serial clock set to System clock/64. - Slave Select disabled. - Input data sampled in the middle of interval. - Clock idle state low. - Serial output data changes on transition from idle clock state to active clock state
Parameters	None.
Returns	Nothing.
Requires	MCU must have the SPI1 module.
Example	`// Initialize the SPI1 module with default settings` `SPI1_Init();`
Notes	SPI library routines require you to specify the module you want to use. To select the desired SPI module, simply change the letter **x** in the routine prototype for a number from **1** to **3**. Number of SPI modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library. Switching between the SPI modules in the SPI library is done by the SPI_Set_Active function (both SPI modules have to be previously initialized).

SPIx_Init_Advanced

Prototype	`procedure SPIx_Init_Advanced(master_mode, data_mode, clock_divider, slave_select, data_sample, clock_idle, edge: word);`
Description	Configures and initializes the SPI module with user defined settings.
Parameters	Parameters `master_mode, data_mode, clock_divider, slave_select, data_sample, clock_idle` and determine the working mode for SPI.

The `master_mode` parameter determines the working mode for SPI module.

Master/Slave mode	
Description	**Predefined library const**
Master mode	_SPI_MASTER
Slave mode	_SPI_SLAVE

The parameter `data_mode` the data length mode, which can be 8-bits (per transmitions cycle), 16-bits or 32-bits.

Data Length Mode	
Description	**Predefined library const**
32-bit mode	_SPI_32_BIT
16-bit mode	_SPI_16_BIT
8-bit mode	_SPI_8_BIT

The parameter `clock_divider` determines the value of the SPI clock speed. Used only in the Master Mode.

Clock Divider Value
Valid values from `2` to `1024`

The parameter `slave_select` determines whether the Slave Select (SS) pin is used in communication. Valid in the Slave Mode only.

Slave Select Enable/Disable	
Description	**Predefined library const**
SS used for the Slave mode	_SPI_SS_ENABLE
SS not used for the Slave mode	_SPI_SS_DISABLE

The parameter `data_sample` determines the sample moment (phase) of input data.

Data Sampling Moment	
Description	**Predefined library const**
Data sampled in the middle of data output time	_SPI_DATA_SAMPLE_MIDDLE
Data sampled at end of data output time	_SPI_DATA_SAMPLE_END

<table>
<tr><td>Parameters</td><td>The parameter <code>clock_idle</code> determines the behaviour of the SPI clock (CLK) line in IDLE phase.
<table>
<tr><th colspan="2">Clock Polarity</th></tr>
<tr><th>Description</th><th>Predefined library const</th></tr>
<tr><td><code>IDLE state is Lo, ACTIVE state is Hi</code></td><td><code>_SPI_CLK_IDLE_LOW</code></td></tr>
<tr><td><code>IDLE state is Hi, ACTIVE state is Lo</code></td><td><code>_SPI_CLK_IDLE_HIGH</code></td></tr>
</table>
The parameter <code>edge</code> determines on which clock edge data is considered to be valid.
<table>
<tr><th colspan="2">Clock Edge</th></tr>
<tr><th>Description</th><th>Predefined library const</th></tr>
<tr><td><code>Data is valid on ACTIVE-to-IDLE transition</code></td><td><code>_SPI_ACTIVE_2_IDLE</code></td></tr>
<tr><td><code>Data is valid on IDLE-to-ACTIVE transition</code></td><td><code>_SPI_IDLE_2_ACTIVE</code></td></tr>
</table></td></tr>
<tr><td>Returns</td><td>Nothing.</td></tr>
<tr><td>Requires</td><td>MCU must have the SPI module.</td></tr>
<tr><td>Example</td><td><pre>// Set SPI1 to the Master Mode, data length is 16-bit, clock = Fcy (no clock scaling), data sampled in the middle of interval, clock IDLE state high and data transmitted at low to high clock edge:
SPI1_Init_Advanced(_SPI_MASTER, _SPI_16_BIT, 1, _SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_HIGH, _SPI_ACTIVE_2_IDLE);</pre></td></tr>
<tr><td>Notes</td><td>SPI library routines require you to specify the module you want to use. To select the desired SPI module, simply change the letter x in the routine prototype for a number from 1 to 3.

Number of SPI modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.</td></tr>
</table>

SPIx_Read

Prototype	`function SPIx_Read(data_out: word): word;`
Description	Reads one word or byte (depending on mode set by init routines) from the SPI bus.
Parameters	- `data_out`: dummy data for clock generation (see device Datasheet for SPI modules implementation details)
Returns	Received data.
Requires	Routine requires at least one SPI module. Used SPI module must be initialized before using this function. See the SPIx_Init and SPIx_Init_Advanced routines.

Example	`// read a byte from the SPI bus`
	`var take, buffer : byte;`
	`...`
	`take := SPI1_Read(buffer);`

Notes	SPI library routines require you to specify the module you want to use. To select the desired SPI module, simply change the letter **x** in the routine prototype for a number from **1** to **3**. Number of SPI modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

SPIx_Write

Prototype	`procedure SPIx_Write(data_out : word);`
Description	Writes one word or byte (depending on mode set by init routines) via the SPI bus.
Parameters	- `data_out`: data to be sent
Returns	Nothing.
Requires	Routine requires at least one SPI module. Used SPI module must be initialized before using this function. See the SPIx_Init and SPIx_Init_Advanced routines.

Example	`// write a byte to the SPI bus`
	`var buffer : byte;`
	`...`
	`SPI1_Write(buffer);`

Notes	SPI library routines require you to specify the module you want to use. To select the desired SPI module, simply change the letter **x** in the routine prototype for a number from **1** to **3**. Number of SPI modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

SPI_Set_Active

Prototype	`procedure SPI_Set_Active(read_ptr : ^TSPI_Rd_Ptr; write_ptr : ^TSPI_Wr_Ptr);`
Description	Sets the active SPI module which will be used by the SPIx_Read and SPIx_Write routines.
Parameters	Parameters: - `read_ptr`: SPI1_Read handler - `write_ptr`: SPI1_Write handler
Returns	Nothing.
Requires	Routine is available only for MCUs with multiple SPI modules. Used SPI module must be initialized before using this function. See the SPIx_Init and SPIx_Init_Advanced routines.
Example	`SPI_Set_Active(@SPI1_Read, @SPI1_Write); // Sets the SPI1 module active`
Notes	Number of SPI modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

Library Example

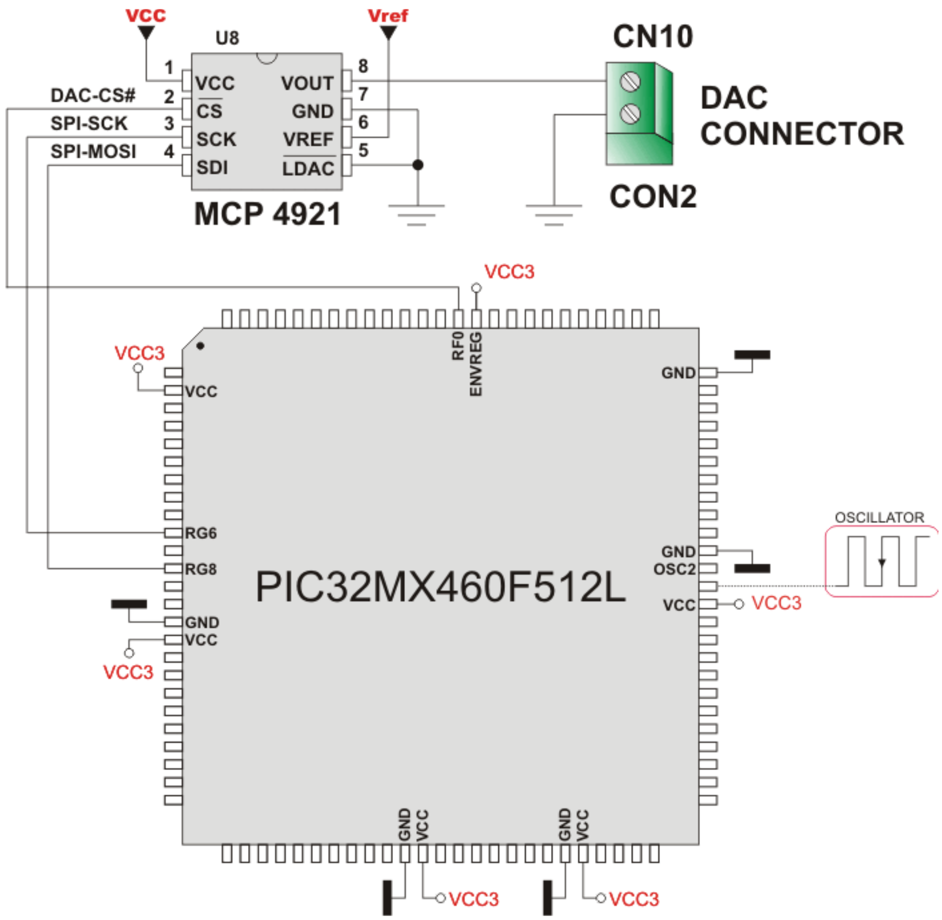
The code demonstrates how to use SPI library functions for communication between SPI2 module of the MCU and MCP4921 DAC chip.

Copy Code To Clipboard

```
program SPI;

// DAC module connections
var Chip_Select : sbit at LATF0_bit;
    Chip_Select_Direction : sbit at TRISF0_bit;
// End DAC module connections

var value : word;

procedure InitMain();
  begin
    TRISB0_bit := 1;                        // Set RB0 pin as input
    TRISB1_bit := 1;                        // Set RB1 pin as input
    Chip_Select := 1;                       // Deselect DAC
    Chip_Select_Direction := 0;             // Set CS# pin as Output
    SPI2_Init();                            // Initialize SPI2 module
  end;

// DAC increments (0..4095) --> output voltage (0..Vref)
procedure DAC_Output( valueDAC : word);
var temp : byte;
  begin
    Chip_Select := 0;                        // Select DAC chip
```

```
    // Send High Byte
    temp := word(valueDAC shr 8) and 0x0F;    // Store valueDAC[11..8] to temp[3..0]
    temp := temp or 0x30;                   // Define DAC setting, see MCP4921 datasheet
    SPI2_Write(temp);                         // Send high byte via SPI

    // Send Low Byte
    temp := valueDAC;                         // Store valueDAC[7..0] to temp[7..0]
    SPI2_Write(temp);                         // Send low byte via SPI

    Chip_Select := 1;                         // Deselect DAC chip
  end;

begin

  CHECON := 0x32;
  AD1PCFG := 0xFFFF;                      // Configure AN pins as digital

  InitMain();                             // Perform main initialization

  value := 2048;                          // When program starts, DAC gives
                                          //   the output in the mid-range

  InitMain();                             // Perform main initialization

  value := 2048;                          // When program starts, DAC gives
                                          //   the output in the mid-range

  while ( TRUE ) do                       // Endless loop
    begin

      if ((RB0_bit) and (value < 4095)) then         // If RA0 button is pressed
        Inc(value)                                   //   increment value
      else
        begin
          if ((RB1_bit) and (value > 0)) then        // If RA1 button is pressed
            Dec(value);                              //   decrement value
        end;

      DAC_Output(value);                             // Send value to DAC chip
      Delay_ms(1);                                   // Slow down key repeat pace
    end;
end.
```

HW Connection



SPI HW connection

SPI Ethernet Library

The `ENC28J60` is a stand-alone Ethernet controller with an industry standard Serial Peripheral Interface (SPI). It is designed to serve as an Ethernet network interface for any controller equipped with SPI.

The `ENC28J60` meets all of the IEEE 802.3 specifications. It incorporates a number of packet filtering schemes to limit incoming packets. It also provides an internal DMA module for fast data throughput and hardware assisted IP checksum calculations. Communication with the host controller is implemented via two interrupt pins and the SPI, with data rates of up to 10 Mb/s. Two dedicated pins are used for LED link and network activity indication.

This library is designed to simplify handling of the underlying hardware (`ENC28J60`). It works with any PIC32 with integrated SPI and more than 4 Kb ROM memory. 38 to 40 MHz clock is recommended to get from 8 to 10 Mhz SPI clock, otherwise PIC32 should be clocked by `ENC28J60` clock output due to its silicon bug in SPI hardware. If you try lower PIC32 clock speed, there might be board hang or miss some requests.

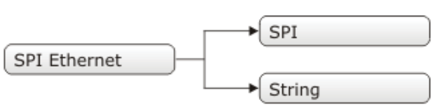
SPI Ethernet library supports:

- IPv4 protocol.
- ARP requests.
- ICMP echo requests.
- UDP requests.
- TCP requests (no stack, no packet reconstruction).
- ARP client with cache.
- DNS client.
- UDP client.
- DHCP client.
- packet fragmentation is **NOT** supported.

Important:

- Global library variable `SPI_Ethernet_userTimerSec` is used to keep track of time for all client implementations (ARP, DNS, UDP and DHCP). It is user responsibility to increment this variable each second in it's code if any of the clients is used.
- For advanced users there is `__EthEnc28j60Private.mpas` unit in Uses folder of the compiler with description of all routines and global variables, relevant to the user, implemented in the SPI Ethernet Library.
- The appropriate hardware SPI module must be initialized before using any of the SPI Ethernet library routines. Refer to SPI Library.
- For MCUs with multiple SPI modules it is possible to initialize them and then switch by using the `SPI_Set_Active()` routine.

Library Dependency Tree



External dependencies of SPI Ethernet Library

The following variables must be defined in all projects using SPI Ethernet Library:	Description:	Example:
`var SPI_Ethernet_CS : sbit; sfr; external;`	ENC28J60 chip select pin.	`var SPI_Ethernet_CS : sbit at LATF1_bit;`
`var SPI_Ethernet_RST : sbit; sfr; external;`	ENC28J60 reset pin.	`var SPI_Ethernet_RST : sbit at LATF0_bit;`
`var SPI_Ethernet_CS_Direction : sbit; sfr; external;`	Direction of the ENC28J60 chip select pin.	`var SPI_Ethernet_CS_Direction : sbit at TRISF1_bit;`
`var SPI_Ethernet_RST_Direction : sbit; sfr; external;`	Direction of the ENC28J60 reset pin.	`var SPI_Ethernet_RST_Direction : sbit at TRISF0_bit;`
The following routines must be defined in all project using SPI Ethernet Library:	**Description:**	**Examples:**
`function SPI_Ethernet_UserTCP(var remoteHost : array[4] of byte, remotePort : word, localPort : word, reqLength : word) var flags: TEthPktFlags) : word;`	TCP request handler.	Refer to the library example at the bottom of this page for code implementation.
`function SPI_Ethernet_UserUDP(var remoteHost : array[4] of byte, remotePort : word, destPort : word, reqLength : word, var flags: TEthPktFlags) : word;`	UDP request handler.	Refer to the library example at the bottom of this page for code implementation.

Library Routines

- SPI_Ethernet_Init
- SPI_Ethernet_Enable
- SPI_Ethernet_Disable
- SPI_Ethernet_doPacket
- SPI_Ethernet_putByte
- SPI_Ethernet_putBytes
- SPI_Ethernet_putString
- SPI_Ethernet_putConstString
- SPI_Ethernet_putConstBytes
- SPI_Ethernet_getByte
- SPI_Ethernet_getBytes
- SPI_Ethernet_UserTCP
- SPI_Ethernet_UserUDP
- SPI_Ethernet_setUserHandlers
- SPI_Ethernet_getIpAddress
- SPI_Ethernet_getGwIpAddress
- SPI_Ethernet_getDnsIpAddress
- SPI_Ethernet_getIpMask
- SPI_Ethernet_confNetwork
- SPI_Ethernet_arpResolve
- SPI_Ethernet_sendUDP
- SPI_Ethernet_dnsResolve
- SPI_Ethernet_initDHCP
- SPI_Ethernet_doDHCPLeaseTime
- SPI_Ethernet_renewDHCP

SPI_Ethernet_Init

Prototype	`procedure SPI_Ethernet_Init(mac: ^byte; ip: ^byte; fullDuplex: byte);`
Description	This is MAC module routine. It initializes `ENC28J60` controller. This function is internaly splited into 2 parts to help linker when coming short of memory. `ENC28J60` controller settings (parameters not mentioned here are set to default): - receive buffer start address : `0x0000`. - receive buffer end address : `0x19AD`. - transmit buffer start address: `0x19AE`. - transmit buffer end address : `0x1FFF`. - RAM buffer read/write pointers in auto-increment mode. - receive filters set to default: CRC + MAC Unicast + MAC Broadcast in OR mode. - flow control with TX and RX pause frames in full duplex mode. - frames are padded to `60` bytes + CRC. - maximum packet size is set to `1518`. - Back-to-Back Inter-Packet Gap: `0x15` in full duplex mode; `0x12` in half duplex mode. - Non-Back-to-Back Inter-Packet Gap: `0x0012` in full duplex mode; `0x0C12` in half duplex mode. - Collision window is set to `63` in half duplex mode to accomodate some `ENC28J60` revisions silicon bugs. - CLKOUT output is disabled to reduce EMI generation. - half duplex loopback disabled. - LED configuration: default (LEDA-link status, LEDB-link activity).

Parameters	- `mac`: RAM buffer containing valid MAC address. - `ip`: RAM buffer containing valid IP address. - `fullDuplex`: ethernet duplex mode switch. Valid values: `0` (half duplex mode) and `1` (full duplex mode).
Returns	Nothing.
Requires	Global variables: - `SPI_Ethernet_CS`: Chip Select line - `SPI_Ethernet_CS_Direction`: Direction of the Chip Select pin - `SPI_Ethernet_RST`: Reset line - `SPI_Ethernet_RST_Direction`: Direction of the Reset pin must be defined before using this function. The SPI module needs to be initialized. See the SPIx_Init and SPIx_Init_Advanced routines.
Example	see code below
Notes	None.

```
// SPI Ethernet module connections
var SPI_Ethernet_RST : sbit at RF0_bit;
var SPI_Ethernet_CS : sbit at RF1_bit;
var SPI_Ethernet_RST_Direction : sbit at TRISF0_bit;
var SPI_Ethernet_CS_Direction : sbit at TRISF1_bit;

const SPI_Ethernet_HALFDUPLEX = 0;
const SPI_Ethernet_FULLDUPLEX = 1;

var
  myMacAddr : array[6] of byte; // my MAC address
  myIpAddr  : array[4] of byte; // my IP addr
  ...
  myMacAddr[0] := 0x00;
  myMacAddr[1] := 0x14;
  myMacAddr[2] := 0xA5;
  myMacAddr[3] := 0x76;
  myMacAddr[4] := 0x19;
  myMacAddr[5] := 0x3F;

  myIpAddr[0]  := 192;
  myIpAddr[1]  := 168;
  myIpAddr[2]  := 1;
  myIpAddr[3]  := 60;

  SPI1_Init();
  SPI_Ethernet_Init(myMacAddr, myIpAddr, SPI_Ethernet_FULLDUPLEX);
```

SPI_Ethernet_Enable

<table>
<tr><td>Prototype</td><td><code>procedure SPI_Ethernet_Enable(enFlt : byte);</code></td></tr>
<tr><td>Description</td><td>This is MAC module routine. This routine enables appropriate network traffic on the ENC28J60 module by the means of it's receive filters (unicast, multicast, broadcast, crc). Specific type of network traffic will be enabled if a corresponding bit of this routine's input parameter is set. Therefore, more than one type of network traffic can be enabled at the same time. For this purpose, predefined library constants (see the table below) can be ORed to form appropriate input value.

Advanced filtering available in the ENC28J60 module such as Pattern Match, Magic Packet and Hash Table can not be enabled by this routine. Additionaly, all filters, except CRC, enabled with this routine will work in OR mode, which means that packet will be received if any of the enabled filters accepts it.

This routine will change receive filter configuration on-the-fly. It will not, in any way, mess with enabling/disabling receive/transmit logic or any other part of the ENC28J60 module. The ENC28J60 module should be properly cofigured by the means of SPI_Ethernet_Init routine.</td></tr>
<tr><td>Parameters</td><td>- enFlt: network traffic/receive filter flags. Each bit corresponds to the appropriate network traffic/receive filter:
<table>
<tr><th>Bit</th><th>Mask</th><th>Description</th><th>Predefined library const</th></tr>
<tr><td>0</td><td>0x01</td><td>MAC Broadcast traffic/receive filter flag. When set, MAC broadcast traffic will be enabled.</td><td>_SPI_Ethernet_BROADCAST</td></tr>
<tr><td>1</td><td>0x02</td><td>MAC Multicast traffic/receive filter flag. When set, MAC multicast traffic will be enabled.</td><td>_SPI_Ethernet_MULTICAST</td></tr>
<tr><td>2</td><td>0x04</td><td>not used</td><td>none</td></tr>
<tr><td>3</td><td>0x08</td><td>not used</td><td>none</td></tr>
<tr><td>4</td><td>0x10</td><td>not used</td><td>none</td></tr>
<tr><td>5</td><td>0x20</td><td>CRC check flag. When set, packets with invalid CRC field will be discarded.</td><td>_SPI_Ethernet_CRC</td></tr>
<tr><td>6</td><td>0x40</td><td>not used</td><td>none</td></tr>
<tr><td>7</td><td>0x80</td><td>MAC Unicast traffic/receive filter flag. When set, MAC unicast traffic will be enabled.</td><td>_SPI_Ethernet_UNICAST</td></tr>
</table></td></tr>
<tr><td>Returns</td><td>Nothing.</td></tr>
<tr><td>Requires</td><td>Ethernet module has to be initialized. See SPI_Ethernet_Init.</td></tr>
<tr><td>Example</td><td><code>SPI_Ethernet_Enable(_SPI_Ethernet_CRC or _SPI_Ethernet_UNICAST); // enable CRC checking and Unicast traffic</code></td></tr>
<tr><td>Notes</td><td>Advanced filtering available in the ENC28J60 module such as Pattern Match, Magic Packet and Hash Table can not be enabled by this routine. Additionaly, all filters, except CRC, enabled with this routine will work in OR mode, which means that packet will be received if any of the enabled filters accepts it.

This routine will change receive filter configuration on-the-fly. It will not, in any way, mess with enabling/disabling receive/transmit logic or any other part of the ENC28J60 module. The ENC28J60 module should be properly cofigured by the means of SPI_Ethernet_Init routine.</td></tr>
</table>

SPI_Ethernet_Disable

<table>
<tr><td>Prototype</td><td><code>procedure SPI_Ethernet_Disable(disFlt : byte);</code></td></tr>
<tr><td>Description</td><td>This is MAC module routine. This routine disables appropriate network traffic on the <code>ENC28J60</code> module by the means of it's receive filters (unicast, multicast, broadcast, crc). Specific type of network traffic will be disabled if a corresponding bit of this routine's input parameter is set. Therefore, more than one type of network traffic can be disabled at the same time. For this purpose, predefined library constants (see the table below) can be ORed to form appropriate input value.</td></tr>
<tr><td>Parameters</td><td>- <code>disFlt</code>: network traffic/receive filter flags. Each bit corresponds to the appropriate network traffic/receive filter:
<table>
<tr><th>Bit</th><th>Mask</th><th>Description</th><th>Predefined library const</th></tr>
<tr><td>0</td><td><code>0x01</code></td><td>MAC Broadcast traffic/receive filter flag. When set, MAC broadcast traffic will be disabled.</td><td><code>_SPI_Ethernet_BROADCAST</code></td></tr>
<tr><td>1</td><td><code>0x02</code></td><td>MAC Multicast traffic/receive filter flag. When set, MAC multicast traffic will be disabled.</td><td><code>_SPI_Ethernet_MULTICAST</code></td></tr>
<tr><td>2</td><td><code>0x04</code></td><td>not used</td><td>none</td></tr>
<tr><td>3</td><td><code>0x08</code></td><td>not used</td><td>none</td></tr>
<tr><td>4</td><td><code>0x10</code></td><td>not used</td><td>none</td></tr>
<tr><td>5</td><td><code>0x20</code></td><td>CRC check flag. When set, CRC check will be disabled and packets with invalid CRC field will be accepted.</td><td><code>_SPI_Ethernet_CRC</code></td></tr>
<tr><td>6</td><td><code>0x40</code></td><td>not used</td><td>none</td></tr>
<tr><td>7</td><td><code>0x80</code></td><td>MAC Unicast traffic/receive filter flag. When set, MAC unicast traffic will be disabled.</td><td><code>_SPI_Ethernet_UNICAST</code></td></tr>
</table></td></tr>
<tr><td>Returns</td><td>Nothing.</td></tr>
<tr><td>Requires</td><td>Ethernet module has to be initialized. See SPI_Ethernet_Init.</td></tr>
<tr><td>Example</td><td><code>SPI_Ethernet_Disable(_SPI_Ethernet_CRC | _SPI_Ethernet_UNICAST); // disable CRC checking and Unicast traffic</code></td></tr>
<tr><td>Notes</td><td>Advanced filtering available in the <code>ENC28J60</code> module such as <code>Pattern Match, Magic Packet</code> and <code>Hash Table</code> can not be disabled by this routine.

This routine will change receive filter configuration on-the-fly. It will not, in any way, mess with enabling/disabling receive/transmit logic or any other part of the <code>ENC28J60</code> module. The <code>ENC28J60</code> module should be properly cofigured by the means of SPI_Ethernet_Init routine.</td></tr>
</table>

SPI_Ethernet_doPacket

Prototype	`function SPI_Ethernet_doPacket() : byte;`
Description	This is MAC module routine. It processes next received packet if such exists. Packets are processed in the following manner: - ARP & ICMP requests are replied automatically. - upon TCP request the SPI_Ethernet_UserTCP function is called for further processing. - upon UDP request the SPI_Ethernet_UserUDP function is called for further processing.
Parameters	None.
Returns	- `0` - upon successful packet processing (zero packets received or received packet processed successfully). - `1` - upon reception error or receive buffer corruption. `ENC28J60` controller needs to be restarted. - `2` - received packet was not sent to us (not our IP, nor IP broadcast address). - `3` - received IP packet was not IPv4. - `4` - received packet was of type unknown to the library.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	<pre>while true do
	  begin
	    ...
	    SPI_Ethernet_doPacket(); // process received packets
	    ...
	  end;</pre>

Notes	`SPI_Ethernet_doPacket` must be called as often as possible in user's code.

SPI_Ethernet_putByte

Prototype	`procedure SPI_Ethernet_putByte(v : byte);`
Description	This is MAC module routine. It stores one byte to address pointed by the current `ENC28J60` write pointer (`EWRPT`).
Parameters	- `v`: value to store
Returns	Nothing.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	<pre>var
	  data : byte;
	  ...
	  SPI_Ethernet_putByte(data); // put an byte into ENC28J60 buffer</pre>

Notes	None.

SPI_Ethernet_putBytes

Prototype	`procedure SPI_Ethernet_putBytes(ptr : ^byte; n : word);`
Description	This is MAC module routine. It stores requested number of bytes into `ENC28J60` RAM starting from current ENC28J60 write pointer (`EWRPT`) location.
Parameters	- `ptr`: RAM buffer containing bytes to be written into `ENC28J60` RAM. - `n`: number of bytes to be written.
Returns	Nothing.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  buffer : array[17] of byte;`
	`  ...`
	`  buffer := 'mikroElektronika';`
	`  ...`
	`   SPI_Ethernet_putBytes(buffer, 16); // put an RAM array into ENC28J60 buffer`

Notes	None.

SPI_Ethernet_putConstBytes

Prototype	`procedure SPI_Ethernet_putConstBytes(const ptr : ^byte; n : word);`
Description	This is MAC module routine. It stores requested number of const bytes into `ENC28J60` RAM starting from current `ENC28J60` write pointer (EWRPT) location.
Parameters	- `ptr`: const buffer containing bytes to be written into `ENC28J60` RAM. - `n`: number of bytes to be written.
Returns	Nothing.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`const`
	`  buffer : array[17] of byte;`
	`  ...`
	`  buffer := 'mikroElektronika';`
	`  ...`
	`  SPI_Ethernet_putConstBytes(buffer, 16); // put a const array into ENC28J60 buffer`

Notes	None.

SPI_Ethernet_putString

Prototype	`function SPI_Ethernet_putString(ptr : ^byte) : word;`
Description	This is MAC module routine. It stores whole string (excluding null termination) into `ENC28J60` RAM starting from current `ENC28J60` write pointer (`EWRPT`) location.
Parameters	- `ptr`: string to be written into `ENC28J60` RAM.
Returns	Number of bytes written into `ENC28J60` RAM.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  buffer : string[16];`
	`  ...`
	`  buffer := 'mikroElektronika';`
	`  ...`
	`  SPI_Ethernet_putString(buffer); // put a RAM string into ENC28J60 buffer`

Notes	None.

SPI_Ethernet_putConstString

Prototype	`function SPI_Ethernet_putConstString(const ptr : ^byte) : word;`
Description	This is MAC module routine. It stores whole const string (excluding null termination) into `ENC28J60` RAM starting from current `ENC28J60` write pointer (EWRPT) location.
Parameters	- `ptr`: const string to be written into `ENC28J60` RAM.
Returns	Number of bytes written into `ENC28J60` RAM.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`const`
	`  buffer : string[16];`
	`  ...`
	`  buffer := 'mikroElektronika';`
	`  ...`
	`  SPI_Ethernet_putConstString(buffer); // put a const string into ENC28J60 buffer`

Notes	None.

SPI_Ethernet_getByte

Prototype	`function SPI_Ethernet_getByte() : byte;`
Description	This is MAC module routine. It fetches a byte from address pointed to by current `ENC28J60` read pointer (`ERDPT`).
Parameters	None.
Returns	Byte read from `ENC28J60` RAM.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  buffer : byte;`
	`  ...`
	`  buffer := SPI_Ethernet_getByte(); // read a byte from ENC28J60 buffer`

Notes	None.

SPI_Ethernet_getBytes

Prototype	`procedure SPI_Ethernet_getBytes(ptr : ^byte; addr : word; n : word);`
Description	This is MAC module routine. It fetches equested number of bytes from `ENC28J60` RAM starting from given address. If value of `0xFFFF` is passed as the address parameter, the reading will start from current `ENC28J60` read pointer (`ERDPT`) location.
Parameters	- `ptr`: buffer for storing bytes read from `ENC28J60` RAM. - `addr`: `ENC28J60` RAM start address. Valid values: `0..8192`. - `n`: number of bytes to be read.
Returns	Nothing.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  buffer: array[16] of byte;`
	`  ...`
	`  SPI_Ethernet_getBytes(buffer, 0x100, 16); // read 16 bytes, starting from address 0x100`

Notes	None.

SPI_Ethernet_UserTCP

Prototype	`function SPI_Ethernet_UserTCP(var remoteHost : array[4] of byte; remotePort, localPort, reqLength : word; var flags: TEthPktFlags) : word;`
Description	This is TCP module routine. It is internally called by the library. The user accesses to the TCP request by using some of the SPI_Ethernet_get routines. The user puts data in the transmit buffer by using some of the SPI_Ethernet_put routines. The function must return the length in bytes of the TCP reply, or 0 if there is nothing to transmit. If there is no need to reply to the TCP requests, just define this function with return(0) as a single statement.

Parameters	- `remoteHost`: client's IP address.
	- `remotePort`: client's TCP port.
	- `localPort`: port to which the request is sent.
	- `reqLength`: TCP request data field length.
	- `flags`: structure consisted of two bit fields :
	
	Copy Code To Clipboard
	
	`type TEthPktFlags = record`
	`  canCloseTCP: boolean;  // flag which closes socket`
	`  isBroadcast: boolean;  // flag which denotes that the IP package has been received via subnet broadcast address`
	`end;`

Returns	- `0` - there should not be a reply to the request. - Length of TCP reply data field - otherwise.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.
Example	This function is internally called by the library and should not be called by the user's code.
Notes	The function source code is provided with appropriate example projects. The code should be adjusted by the user to achieve desired reply.

SPI_Ethernet_UserUDP

Prototype	`function SPI_Ethernet_UserUDP(var remoteHost : array[4] of byte; remotePort, destPort, reqLength : word; var flags: TEthPktFlags) : word;`
Description	This is UDP module routine. It is internally called by the library. The user accesses to the UDP request by using some of the SPI_Ethernet_get routines. The user puts data in the transmit buffer by using some of the SPI_Ethernet_put routines. The function must return the length in bytes of the UDP reply, or 0 if nothing to transmit. If you don't need to reply to the UDP requests, just define this function with a return(0) as single statement.

Parameters	- `remoteHost`: client's IP address.
	- `remotePort`: client's port.
	- `localPort`: port to which the request is sent.
	- `reqLength`: UDP request data field length.
	- `flags`: structure consisted of two bit fields :
	
	Copy Code To Clipboard
	
	`type TEthPktFlags = record`
	`  canCloseTCP: boolean;  // flag which closes socket (not relevant to UDP)`
	`  isBroadcast: boolean;  // flag which denotes that the IP package has been received via subnet broadcast address`
	`end;`

Returns	- `0` - there should not be a reply to the request. - Length of UDP reply data field - otherwise.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.
Example	This function is internally called by the library and should not be called by the user's code.
Notes	The function source code is provided with appropriate example projects. The code should be adjusted by the user to achieve desired reply.

SPI_Ethernet_setUserHandlers

Prototype	`procedure SPI_Ethernet_setUserHandlers(TCPHandler : ^TSPI_Ethernet_UserTCP; UDPHandler : ^TSPI_Ethernet_UserUDP);`
Description	Sets pointers to User TCP and UDP handler function implementations, which are automatically called by SPI Ethernet library.
Parameters	- `TCPHandler`: TCP request handler - `UDPHandler`: UDP request handler.
Returns	Nothing.
Requires	SPI_Ethernet_UserTCP and SPI_Ethernet_UserUDP have to be previously defined.
Example	`SPI_Ethernet_setUserHandlers(@SPI_Ethernet_UserTCP, @SPI_Ethernet_UserUDP);`
Notes	Since all libraries are built for SSA, SSA restrictions regarding function pointers dictate that modules that use SPI_Ethernet_setUserHandlers must also be built for SSA.

SPI_Ethernet_getIpAddress

Prototype	`function SPI_Ethernet_getIpAddress() : word;`
Description	This routine should be used when DHCP server is present on the network to fetch assigned IP address.
Parameters	None.
Returns	Pointer to the global variable holding IP address.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  ipAddr : array[4] of byte;  // user IP address buffer`
	`  ...`
	`  memcpy(ipAddr, SPI_Ethernet_getIpAddress(), 4); // fetch IP address`

Notes	User should always copy the IP address from the RAM location returned by this routine into it's own IP address buffer. These locations should not be altered by the user in any case!

Ethernet_getGwIpAddress

Prototype	`function SPI_Ethernet_getGwIpAddress() : word;`
Description	This routine should be used when DHCP server is present on the network to fetch assigned gateway IP address.
Parameters	None.
Returns	Pointer to the global variable holding gateway IP address.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  gwIpAddr : array[4] of byte;  // user gateway IP address buffer`
	`  ...`
	`  memcpy(gwIpAddr, SPI_Ethernet_getGwIpAddress(), 4); // fetch gateway IP address`

Notes	User should always copy the IP address from the RAM location returned by this routine into it's own gateway IP address buffer. These locations should not be altered by the user in any case!

SPI_Ethernet_getDnsIpAddress

Prototype	`function SPI_Ethernet_getDnsIpAddress() : word;`
Description	This routine should be used when DHCP server is present on the network to fetch assigned DNS IP address.
Parameters	None.
Returns	Pointer to the global variable holding DNS IP address.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  dnsIpAddr : array[4] of byte;  // user DNS IP address buffer`
	`  ...`
	`  memcpy(dnsIpAddr, SPI_Ethernet_getDnsIpAddress(), 4); // fetch DNS server address`

Notes	User should always copy the IP address from the RAM location returned by this routine into it's own DNS IP address buffer. These locations should not be altered by the user in any case!

SPI_Ethernet_getIpMask

Prototype	`function SPI_Ethernet_getIpMask() : word;`
Description	This routine should be used when DHCP server is present on the network to fetch assigned IP subnet mask.
Parameters	None.
Returns	Pointer to the global variable holding IP subnet mask.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  IpMask : array[4] of byte;  // user IP subnet mask buffer`
	`  ...`
	`  memcpy(IpMask, SPI_Ethernet_getIpMask(), 4); // fetch IP subnet mask`

Notes	User should always copy the IP address from the RAM location returned by this routine into it's own IP subnet mask buffer. These locations should not be altered by the user in any case!

SPI_Ethernet_confNetwork

Prototype	`procedure SPI_Ethernet_confNetwork(var ipMask, gwIpAddr, dnsIpAddr : array[4] of byte);`
Description	Configures network parameters (IP subnet mask, gateway IP address, DNS IP address) when DHCP is not used.
Parameters	- `ipMask`: IP subnet mask. - `gwIpAddr` gateway IP address. - `dnsIpAddr`: DNS IP address.
Returns	Nothing.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  ipMask    : array[4] of byte;  // network mask (for example : 255.255.255.0)`
	`  gwIpAddr  : array[4] of byte;  // gateway (router) IP address`
	`  dnsIpAddr : array[4] of byte;  // DNS server IP address`
	`  ...`
	`  gwIpAddr[0]  := 192;`
	`  gwIpAddr[1]  := 168;`
	`  gwIpAddr[2]  := 20;`
	`  gwIpAddr[3]  := 6;`
	
	`  dnsIpAddr[0] := 192;`
	`  dnsIpAddr[1] := 168;`
	`  dnsIpAddr[2] := 20;`
	`  dnsIpAddr[3] := 100;`
	
	`  ipMask[0]    := 255;`
	`  ipMask[1]    := 255;`
	`  ipMask[2]    := 255;`
	`  ipMask[3]    := 0;`
	`  ...`
	`  SPI_Ethernet_confNetwork(ipMask, gwIpAddr, dnsIpAddr); // set network configuration parameters`

Notes	The above mentioned network parameters should be set by this routine only if DHCP module is not used. Otherwise DHCP will override these settings.

SPI_Ethernet_arpResolve

Prototype	`function SPI_Ethernet_arpResolve(var ip : array[4] of byte; tmax : byte) : word;`
Description	This is ARP module routine. It sends an ARP request for given IP address and waits for ARP reply. If the requested IP address was resolved, an ARP cash entry is used for storing the configuration. ARP cash can store up to 3 entries. For ARP cash structure refer to "`eth_enc28j60LibDef.h`" header file in the compiler's Uses folder.
Parameters	- `ip`: IP address to be resolved. - `tmax`: time in seconds to wait for an reply.
Returns	- MAC address behind the IP address - the requested IP address was resolved. - `0` - otherwise.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  IpAddr : array[4] of byte;  // IP address`
	`  ...`
	`  IpAddr[0] := 192;`
	`  IpAddr[0] := 168;`
	`  IpAddr[0] := 1;`
	`  IpAddr[0] := 1;`
	`  ...`
	`  SPI_Ethernet_arpResolve(IpAddr, 5); // get MAC address behind the above IP address, wait 5 secs for the response`

Notes	The Ethernet services are not stopped while this routine waits for ARP reply. The incoming packets will be processed normaly during this time.

SPI_Ethernet_sendUDP

Prototype	`function SPI_Ethernet_sendUDP(var destIP : array[4] of byte; sourcePort, destPort : word; pkt : ^byte; pktLen : word) : byte;`
Description	This is UDP module routine. It sends an UDP packet on the network.
Parameters	- `destIP`: remote host IP address. - `sourcePort`: local UDP source port number. - `destPort`: destination UDP port number. - `pkt`: packet to transmit. - `pktLen`: length in bytes of packet to transmit.
Returns	- `1` - UDP packet was sent successfully. - `0` - otherwise.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  IpAddr : array[4] of byte;  // remote IP address`
	`  ...`
	`  IpAddr[0] := 192;`
	`  IpAddr[0] := 168;`
	`  IpAddr[0] := 1;`
	`  IpAddr[0] := 1;`
	`  ...`
	`  SPI_Ethernet_sendUDP(IpAddr, 10001, 10001, 'Hello', 5); // send Hello message to the above IP address, from UDP port 10001 to UDP port 10001`

Notes	None.

SPI_Ethernet_dnsResolve

Prototype	`function SPI_Ethernet_dnsResolve(var host : string; tmax : byte) : word;`
Description	This is DNS module routine. It sends an DNS request for given host name and waits for DNS reply. If the requested host name was resolved, it's IP address is stored in library global variable and a pointer containing this address is returned by the routine. UDP port 53 is used as DNS port.
Parameters	- `host`: host name to be resolved. - `tmax`: time in seconds to wait for an reply.
Returns	- pointer to the location holding the IP address - the requested host name was resolved. - `0` - otherwise.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`var`
	`  remoteHostIpAddr : array[4] of byte;  // user host IP address buffer`
	`  ...`
	`  // SNTP server:`
	`   //  Zurich,  Switzerland:  Integrated  Systems  Lab,  Swiss  Fed.  Inst.  of Technology`
	`  // 129.132.2.21: swisstime.ethz.ch`
	`  // Service Area: Switzerland and Europe`
	`  memcpy(remoteHostIpAddr, SPI_Ethernet_dnsResolve('swisstime.ethz.ch', 5), 4);`

Notes	The Ethernet services are not stopped while this routine waits for DNS reply. The incoming packets will be processed normaly during this time. User should always copy the IP address from the RAM location returned by this routine into it's own resolved host IP address buffer. These locations should not be altered by the user in any case!

SPI_Ethernet_initDHCP

Prototype	`function SPI_Ethernet_initDHCP(tmax : byte) : byte;`
Description	This is DHCP module routine. It sends an DHCP request for network parameters (IP, gateway, DNS addresses and IP subnet mask) and waits for DHCP reply. If the requested parameters were obtained successfully, their values are stored into the library global variables. These parameters can be fetched by using appropriate library IP get routines: - SPI_Ethernet_getIpAddress - fetch IP address. - SPI_Ethernet_getGwIpAddress - fetch gateway IP address. - SPI_Ethernet_getDnsIpAddress - fetch DNS IP address. - SPI_Ethernet_getIpMask - fetch IP subnet mask. UDP port `68` is used as DHCP client port and UDP port `67` is used as DHCP server port.
Parameters	- `tmax`: time in seconds to wait for an reply.
Returns	- `1` - network parameters were obtained successfully. - `0` - otherwise.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`...`
	`SPI_Ethernet_initDHCP(5); // get network configuration from DHCP server, wait 5 sec for the response`
	`...`

Notes	The Ethernet services are not stopped while this routine waits for DNS reply. The incoming packets will be processed normaly during this time. When DHCP module is used, global library variable `SPI_Ethernet_userTimerSec` is used to keep track of time. It is user responsibility to increment this variable each second in it's code.

SPI_Ethernet_doDHCPLeaseTime

Prototype	`function SPI_Ethernet_doDHCPLeaseTime() : byte;`
Description	This is DHCP module routine. It takes care of IP address lease time by decrementing the global lease time library counter. When this time expires, it's time to contact DHCP server and renew the lease.
Parameters	None
Returns	- `0` - lease time has not expired yet. - `1` - lease time has expired, it's time to renew it.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	`while true do`
	`  begin`
	`    ...`
	`    if (SPI_Ethernet_doDHCPLeaseTime() <> 0) then`
	`      begin`
	`        ... // it's time to renew the IP address lease`
	`      end;`
	`  end;`

Notes	None.

SPI_Ethernet_renewDHCP

Prototype	`function SPI_Ethernet_renewDHCP(tmax : byte) : byte;`
Description	This is DHCP module routine. It sends IP address lease time renewal request to DHCP server.
Parameters	- `tmax`: time in seconds to wait for an reply.
Returns	- `1` - upon success (lease time was renewed). - `0` - otherwise (renewal request timed out).
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.

Example	<pre>while true do
	  begin
	    ...
	    if (SPI_Ethernet_doDHCPLeaseTime() <> 0) then
	      begin
	        SPI_Ethernet_renewDHCP(5); // it's time to renew the IP address
	lease, with 5 secs for a reply
	      end;
	    ...
	  end;</pre>

Notes	None.

Library Example

This code shows how to use the Ethernet mini library:

- the board will reply to ARP & ICMP echo requests
- the board will reply to UDP requests on any port:
  - returns the request in upper char with a header made of remote host IP & port number
- the board will reply to HTTP requests on port 80, GET method with pathnames:
  - / will return the HTML main page
  - /s will return board status as text string
  - /t0 ... /t7 will toggle RD0 to RD7 bit and return HTML main page
  - all other requests return also HTML main page.

Copy Code To Clipboard

```
program HTTP_Demo;

{***********************************
 * RAM variables
 *}

// mE ehternet NIC pinout
var
  SPI_Ethernet_Rst : sbit at LATF0_bit;  // for writing to output pin always use latch
  SPI_Ethernet_CS  : sbit at LATF1_bit;  // for writing to output pin always use latch
  SPI_Ethernet_Rst_Direction : sbit at TRISF0_bit;
  SPI_Ethernet_CS_Direction  : sbit at TRISF1_bit;
// end ethernet NIC definitions

var myMacAddr   : array[6] of byte; // my MAC address
    myIpAddr    : array[4] of byte; // my IP address
    gwIpAddr    : array[4] of byte; // gateway (router) IP address
    ipMask      : array[4] of byte; // network mask (for example : 255.255.255.0)
    dnsIpAddr   : array[4] of byte; // DNS server IP address

{************************************************************
 * ROM constant strings
 *}
const  httpHeader  :  string[30]  =  'HTTP/1.1  200  OK'+#10+'Content-type:  ';   //  HTTP
header
const httpMimeTypeHTML : string[11]   = 'text/html'+#10+#10;         // HTML MIME type
const httpMimeTypeScript : string[12] = 'text/plain'+#10+#10;        // TEXT MIME type
const httpMethod : string[5] = 'GET /';
{*
 * web page, splited into 2 parts :
 * when coming short of ROM, fragmented data is handled more efficiently by linker
 *
  *  this  HTML  page  calls  the  boards  to  get  its  status,  and  builds  itself  with
javascript
 *}
const indexPage : string[761] =
```

```
                    '<meta http-equiv="refresh" content="3;url=http://192.168.20.60">' +
                    '<HTML><HEAD></HEAD><BODY>'+
                    '<h1>PIC32 + ENC28J60 Mini Web Server</h1>'+
                    '<a href=/>Reload</a>'+
                    '<script src=/s></script>'+
                      '<table><tr><td valign=top><table border=1 style="font-size:20px
;font-family: terminal ;">'+
                    '<tr><th colspan=2>ADC</th></tr>'+
                      '<tr><td>AN0</td><td><script>document.write(AN0)</script></td></
tr>'+
                      '<tr><td>AN1</td><td><script>document.write(AN1)</script></td></
tr>'+
                  '</table></td><td><table border=1 style="font-size:20px ;font-family:
terminal ;">'+
                    '<tr><th colspan=2>PORTB</th></tr>'+
                    '<script>'+
                    'var str,i;'+
                    'str="";'+
                    'for(i=2;i<10;i++)'+
                    '{str+="<tr><td bgcolor=pink>BUTTON #"+i+"</td>";'+
                    'if(PORTB&(1<<i)){str+="<td bgcolor=red>ON";}'+
                    'else {str+="<td bgcolor=#cccccc>OFF";}'+
                    'str+="</td></tr>";}'+
                    'document.write(str);'+
                    '</script>';

const indexPage2 : string[466] =
                    '</table></td><td>'+
                        '<table border=1 style="font-size:20px ;font-family: terminal
;">'+
                    '<tr><th colspan=3>PORTD</th></tr>'+
                    '<script>'+
                    'var str,i;'+
                    'str="";'+
                    'for(i=0;i<8;i++)'+
                    '{str+="<tr><td bgcolor=yellow>LED #"+i+"</td>";'+
                    'if(PORTD&(1<<i)){str+="<td bgcolor=red>ON";}'+
                    'else {str+="<td bgcolor=#cccccc>OFF";}'+
                    'str+="</td><td><a href=/t"+i+">Toggle</a></td></tr>";}'+
                    'document.write(str);'+
                    '</script>'+
                    '</table></td></tr></table>'+
                   'This is HTTP request #<script>document.write(REQ)</script></BODY></
HTML>';

var    getRequest  : array[15] of byte;  // HTTP request buffer
       dyna        : array[30] of char;  // buffer for dynamic response
       httpCounter : word;               // counter of HTTP requests

{*******************************************
 * user defined functions
 *}
```

```
{*
 * this function is called by the library
 * the user accesses to the HTTP request by successive calls to SPI_Ethernet_getByte()
 * the user puts data in the transmit buffer by successive calls to SPI_Ethernet_
putByte()
 * the function must return the length in bytes of the HTTP reply, or 0 if nothing to
transmit
 *
 * if you don't need to reply to HTTP requests,
 * just define this function with a return(0) as single statement
 *
 *}
function SPI_Ethernet_UserTCP(var remoteHost : array[4] of byte;
                                    remotePort, localPort, reqLength : word; var flags:
TEthPktFlags) : word;
  var  i : word;        // my reply length
       bitMask : dword; // for bit mask
       tmp: string[11]; // to copy const array to ram for memcmp
  begin
    result := 0;

    // should we close tcp socket after response is sent?
    // library closes tcp socket by default if canCloseTCP flag is not reset here
    // flags.canCloseTCP := 0; // 0 - do not close socket
                               // otherwise - close socket

    if(localPort <> 80) then  // I listen only to web request on port 80
      begin
        result := 0;
        exit;
      end;

    // get 10 first bytes only of the request, the rest does not matter here
    for i := 0 to 9 do
      getRequest[i] := SPI_Ethernet_getByte();
    getRequest[i] := 0;

    // copy httpMethod to ram for use in memcmp routine
    for i := 0 to 4 do
      tmp[i] := httpMethod[i];

    if(memcmp(@getRequest, @tmp, 5) <> 0) then  // only GET method is supported here
      begin
        result := 0;
        exit;
      end;

    Inc(httpCounter);                          // one more request done

    if(getRequest[5] = 's') then               // if request path name starts with s,
store dynamic data in transmit buffer
      begin
         // the text string replied by this request can be interpreted as javascript
statements
         // by browsers
```

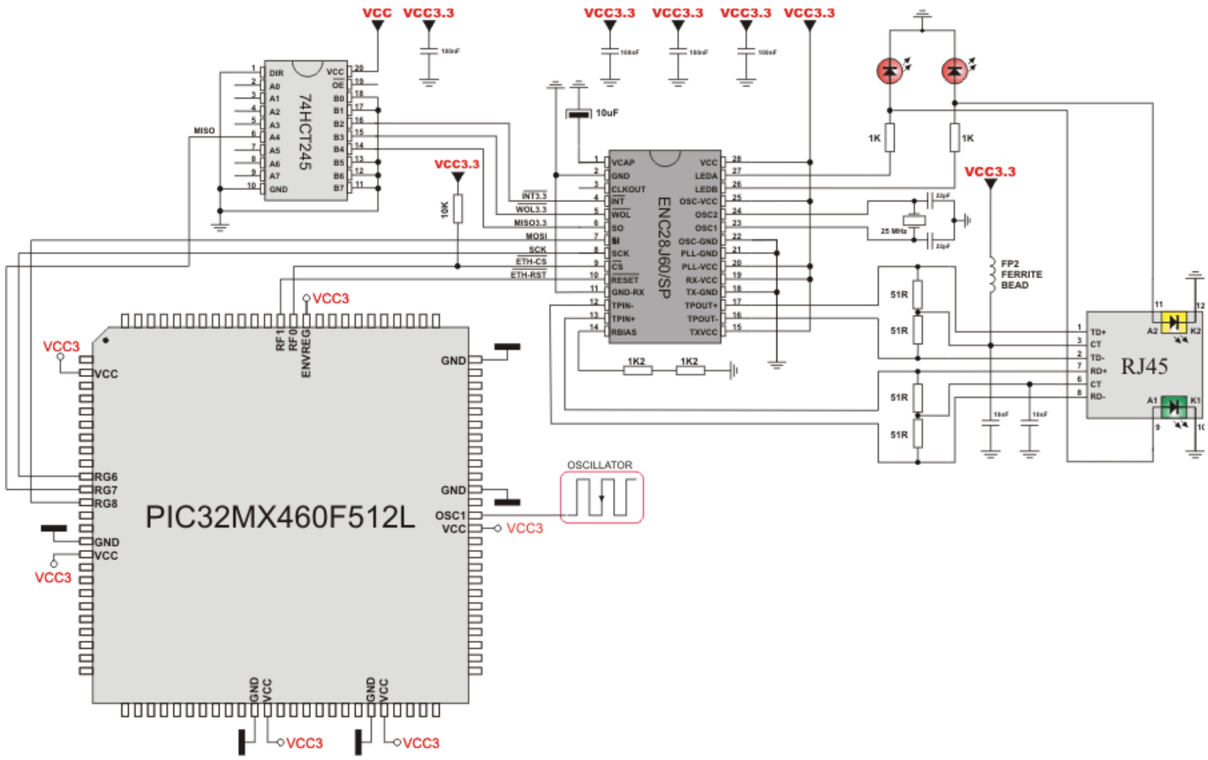
```
         result := SPI_Ethernet_putConstString(@httpHeader);         // HTTP header
         result := result + SPI_Ethernet_putConstString(@httpMimeTypeScript);  // with
text MIME type

        // add AN2 value to reply
        WordToStr(ADC1_Get_Sample(0), dyna);
        tmp := 'var AN0=';
        result := result + SPI_Ethernet_putString(@tmp);
        result := result + SPI_Ethernet_putString(@dyna);
        tmp := ';';
        result := result + SPI_Ethernet_putString(@tmp);

        // add AN3 value to reply
        WordToStr(ADC1_Get_Sample(1), dyna);
        tmp := 'var AN1=';
        result := result + SPI_Ethernet_putString(@tmp);
        result := result + SPI_Ethernet_putString(@dyna);
        tmp := ';';
        result := result + SPI_Ethernet_putString(@tmp);

        // add PORTB value (buttons) to reply
        tmp := 'var PORTB= ';
        result := result + SPI_Ethernet_putString(@tmp);
        WordToStr(PORTB, dyna);
        result := result + SPI_Ethernet_putString(@dyna);
        tmp := ';';
        result := result + SPI_Ethernet_putString(@tmp);

        // add PORTD value (LEDs) to reply
        tmp := 'var PORTD= ';
        result := result + SPI_Ethernet_putString(@tmp);
        WordToStr(PORTD, dyna);
        result := result + SPI_Ethernet_putString(@dyna);
        tmp := ';';
        result := result + SPI_Ethernet_putString(@tmp);

        // add HTTP requests counter to reply
        WordToStr(httpCounter, dyna);
        tmp := 'var REQ=   ';
        result := result + SPI_Ethernet_putString(@tmp);
        result := result + SPI_Ethernet_putString(@dyna);
        tmp := ';';
        result := result + SPI_Ethernet_putString(@tmp);
      end
    else
      if(getRequest[5] = 't') then                       // if request path name starts
with t, toggle PORTD (LED) bit number that comes after
        begin
          bitMask := 0;
          if(isdigit(getRequest[6]) <> 0) then            // if 0 <= bit number <= 9,
bits 8 & 9 does not exist but does not matter
            begin
              bitMask := getRequest[6] - '0';            // convert ASCII to integer
```

```
              bitMask := 1 shl bitMask;           // create bit mask
              LATD   := PORTD xor bitMask;         // toggle PORTD with xor operator
            end;
        end;

    if(result = 0) then // what do to by default
      begin
        result := SPI_Ethernet_putConstString(@httpHeader);            // HTTP header
        result := result + SPI_Ethernet_putConstString(@httpMimeTypeHTML);  // with HTML
MIME type
        result := result + SPI_Ethernet_putConstString(@indexPage);         // HTML page
first part
        result := result + SPI_Ethernet_putConstString(@indexPage2);        // HTML page
second part
      end;
    // return to the library with the number of bytes to transmit
end;

{*
 * this function is called by the library
 * the user accesses to the UDP request by successive calls to SPI_Ethernet_getByte()
 * the user puts data in the transmit buffer by successive calls to SPI_Ethernet_
putByte()
 * the function must return the length in bytes of the UDP reply, or 0 if nothing to
transmit
 *
 * if you don't need to reply to UDP requests,
 * just define this function with a return(0) as single statement
 *
 *}
function SPI_Ethernet_UserUDP(var remoteHost : array[4] of byte;
                                     remotePort, destPort, reqLength : word; var flags:
TEthPktFlags) : word;
  var tmp : string[5];
  begin
    result := 0;
    // reply is made of the remote host IP address in human readable format
    byteToStr(remoteHost[0], dyna);           // first IP address byte
    dyna[3] := '.';
    byteToStr(remoteHost[1], tmp);            // second
    dyna[4] := tmp[0];
    dyna[5] := tmp[1];
    dyna[6] := tmp[2];
    dyna[7] := '.';
    byteToStr(remoteHost[2], tmp);            // second
    dyna[8] := tmp[0];
    dyna[9] := tmp[1];
    dyna[10] := tmp[2];
    dyna[11] := '.';
    byteToStr(remoteHost[3], tmp);            // second
    dyna[12] := tmp[0];
    dyna[13] := tmp[1];
    dyna[14] := tmp[2];

    dyna[15] := ':';                          // add separator
```

```
    // then remote host port number
    WordToStr(remotePort, tmp);
    dyna[16] := tmp[0];
    dyna[17] := tmp[1];
    dyna[18] := tmp[2];
    dyna[19] := tmp[3];
    dyna[20] := tmp[4];
    dyna[21] := '[';
    WordToStr(destPort, tmp);
    dyna[22] := tmp[0];
    dyna[23] := tmp[1];
    dyna[24] := tmp[2];
    dyna[25] := tmp[3];
    dyna[26] := tmp[4];
    dyna[27] := ']';
    dyna[28] := 0;

    // the total length of the request is the length of the dynamic string plus the text
of the request
    result := 28 + reqLength;

    // puts the dynamic string into the transmit buffer
    SPI_Ethernet_putBytes(@dyna, 28);

    // then puts the request string converted into upper char into the transmit buffer
    while(reqLength <> 0) do
      begin
        SPI_Ethernet_putByte(SPI_Ethernet_getByte());
        reqLength := reqLength - 1;
      end;
    // back to the library with the length of the UDP reply
  end;

begin
  CHECON := 0x32;
  AD1PCFG := 0xFFFC;         // all digital but RB1(AN1) and RB0 (AN0)

  PORTB := 0;
  TRISB := 0xFFFF;           // set PORTB as input for buttons and adc

  PORTD := 0;
  TRISD := 0;                // set PORTD as output,

  ADC1_Init();

  httpCounter := 0;

  // set mac address
  myMacAddr[0] := 0x00;
  myMacAddr[1] := 0x14;
  myMacAddr[2] := 0xA5;
  myMacAddr[3] := 0x76;
  myMacAddr[4] := 0x19;
  myMacAddr[5] := 0x3F;
```

```
  // set IP address
  myIpAddr[0] := 192;
  myIpAddr[1] := 168;
  myIpAddr[2] := 20;
  myIpAddr[3] := 60;

  // set gateway address
  gwIpAddr[0]  := 192;
  gwIpAddr[1]  := 168;
  gwIpAddr[2]  := 20;
  gwIpAddr[3]  := 6;

  // set dns address
  dnsIpAddr[0] := 192;
  dnsIpAddr[1] := 168;
  dnsIpAddr[2] := 20;
  dnsIpAddr[3] := 1;

  // set subnet mask
  ipMask[0]    := 255;
  ipMask[1]    := 255;
  ipMask[2]    := 255;
  ipMask[3]    := 0;

  {*
   * starts ENC28J60 with :
   * reset bit on PORTC.B0
   * CS bit on PORTC.B1
   * my MAC & IP address
   * full duplex
   *}

  SPI2_Init_Advanced(_SPI_MASTER, _SPI_8_BIT, 16,
                     _SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_LOW, _SPI_
IDLE_2_ACTIVE);
  SPI_Ethernet_Init(myMacAddr, myIpAddr, _SPI_Ethernet_FULLDUPLEX);             // init
ethernet module
  SPI_Ethernet_setUserHandlers(@SPI_Ethernet_UserTCP, @SPI_Ethernet_UserUDP);    // set
user handlers

  // dhcp will not be used here, so use preconfigured addresses
  SPI_Ethernet_confNetwork(ipMask, gwIpAddr, dnsIpAddr);
  while true do                   // do forever
    begin
      SPI_Ethernet_doPacket();    // process incoming Ethernet packets

      {*
       * add your stuff here if needed
       * SPI_Ethernet_doPacket() must be called as often as possible
       * otherwise packets could be lost
       *}
    end;
end.
```

HW Connection



SPI Ethernet ENC24J600 Library

The ENC24J600 is a stand-alone Ethernet controller with an industry standard Serial Peripheral Interface (SPI). It is designed to serve as an Ethernet network interface for any controller equipped with SPI.

The ENC24J600 meets all of the IEEE 802.3 specifications applicable to 10Base-T and 100Base-TX Ethernet. It incorporates a number of packet filtering schemes to limit incoming packets. It also provides an internal, 16-bit wide DMA module for fast data throughput and hardware assisted IP checksum calculations. Communication with the host controller is implemented via two interrupt pins and the SPI, with data rates of 10/100 Mb/s. Two dedicated pins are used for LED link and network activity indication.

This library is designed to simplify handling of the underlying hardware (ENC24J600). It works with any PIC32 with integrated SPI and more than 4 Kb ROM memory. 38 to 40 MHz clock is recommended to get from 8 to 10 Mhz SPI clock, otherwise PIC32 should be clocked by ENC24J600 clock output due to its silicon bug in SPI hardware. If you try lower PIC32 clock speed, there might be board hang or miss some requests.

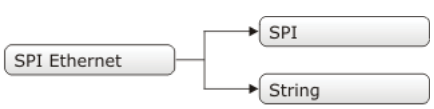
SPI Ethernet ENC24J600 library supports:

- IPv4 protocol.
- ARP requests.
- ICMP echo requests.
- UDP requests.
- TCP requests (no stack, no packet reconstruction).
- ARP client with cache.
- DNS client.
- UDP client.
- DHCP client.
- packet fragmentation is **NOT** supported.

Important:

- Global library variable `SPI_Ethernet_24j600_userTimerSec` is used to keep track of time for all client implementations (ARP, DNS, UDP and DHCP). It is user responsibility to increment this variable each second in it's code if any of the clients is used.
- For advanced users there is `__EthEnc24j600Private.mpas` unit in Uses folder of the compiler with description of all routines and global variables, relevant to the user, implemented in the SPI Ethernet ENC24J600 Library.
- The appropriate hardware SPI module must be initialized before using any of the SPI Ethernet ENC24J600 library routines. Refer to SPI Library.
- For MCUs with multiple SPI modules it is possible to initialize them and then switch by using the `SPI_Set_Active()` routine.

Library Dependency Tree



External dependencies of SPI Ethernet ENC24J600 Library

The following variables must be defined in all projects using SPI Ethernet ENC24J600 Library:	Description:	Example:
`var SPI_Ethernet_24j600_CS : sbit; sfr; external;`	ENC24J600 chip select pin.	`var SPI_Ethernet_24j600_CS : sbit at LATF1_bit;`
`var SPI_Ethernet_24j600_CS_Direction : sbit; sfr; external;`	Direction of the ENC24J600 chip select pin.	`var SPI_Ethernet_24j600_CS_Direction : sbit at TRISF1_bit;`

The following routines must be defined in all project using SPI Ethernet ENC24J600 Library:	Description:	Example:
`function SPI_Ethernet_24j600_UserTCP(var remoteHost : array[4] of byte, remotePort : word, localPort : word, reqLength : word) var flags: TEthj600PktFlags) : word;`	TCP request handler.	Refer to the library example at the bottom of this page for code implementation.
`function SPI_Ethernet_24j600_UserUDP(var remoteHost : array[4] of byte, remotePort : word, destPort : word, reqLength : word) var flags: TEthj600PktFlags) : word;`	UDP request handler.	Refer to the library example at the bottom of this page for code implementation.

Library Routines

- SPI_Ethernet_24j600_Init
- SPI_Ethernet_24j600_Enable
- SPI_Ethernet_24j600_Disable
- SPI_Ethernet_24j600_doPacket
- SPI_Ethernet_24j600_putByte
- SPI_Ethernet_24j600_putBytes
- SPI_Ethernet_24j600_putString
- SPI_Ethernet_24j600_putConstString
- SPI_Ethernet_24j600_putConstBytes
- SPI_Ethernet_24j600_getByte
- SPI_Ethernet_24j600_getBytes
- SPI_Ethernet_24j600_UserTCP
- SPI_Ethernet_24j600_UserUDP
- SPI_Ethernet_24j600_getIpAddress
- SPI_Ethernet_24j600_getGwIpAddress
- SPI_Ethernet_24j600_getDnsIpAddress
- SPI_Ethernet_24j600_getIpMask
- SPI_Ethernet_24j600_confNetwork
- SPI_Ethernet_24j600_arpResolve
- SPI_Ethernet_24j600_sendUDP
- SPI_Ethernet_24j600_dnsResolve
- SPI_Ethernet_24j600_initDHCP
- SPI_Ethernet_24j600_doDHCPLeaseTime
- SPI_Ethernet_24j600_renewDHCP

SPI_Ethernet_24j600_Init

<table>
<tr><td>Prototype</td><td><code>procedure SPI_Ethernet_24j600_Init(mac: ^byte; ip: ^byte; fullDuplex: configuration);</code></td></tr>
<tr><td>Description</td><td>This is MAC module routine. It initializes <code>ENC24J600</code> controller. This function is internaly splited into 2 parts to help linker when coming short of memory.

<code>ENC24J600</code> controller settings (parameters not mentioned here are set to default):

- receive buffer start address : <code>0x0000</code>.
- receive buffer end address : <code>0x19AD</code>.
- transmit buffer start address: <code>0x19AE</code>.
- transmit buffer end address : <code>0x1FFF</code>.
- RAM buffer read/write pointers in auto-increment mode.
- receive filters set to default: CRC + MAC Unicast + MAC Broadcast in OR mode.
- flow control with TX and RX pause frames in full duplex mode.
- frames are padded to <code>60</code> bytes + CRC.
- maximum packet size is set to <code>1518</code>.
- Back-to-Back Inter-Packet Gap: <code>0x15</code> in full duplex mode; <code>0x12</code> in half duplex mode.
- Non-Back-to-Back Inter-Packet Gap: <code>0x0012</code> in full duplex mode; <code>0x0C12</code> in half duplex mode.
- Collision window is set to <code>63</code> in half duplex mode to accomodate some <code>ENC24J600</code> revisions silicon bugs.
- CLKOUT output is disabled to reduce EMI generation.
- half duplex loopback disabled.
- LED configuration: default (LEDA-link status, LEDB-link activity).</td></tr>
<tr><td>Parameters</td><td>- <code>mac</code>: RAM buffer containing valid MAC address.
- <code>ip</code>: RAM buffer containing valid IP address.
- <code>configuration</code>: ethernet negotiation, duplex and speed mode settings. For this purpose, predefined library constants (see the list below) can be combined using logical AND to form appropriate value:

<table>
<tr><th>Description</th><th>Predefined library const</th></tr>
<tr><td>Set Auto-negotiation</td><td><code>SPI_Ethernet_24j600_AUTO_NEGOTIATION</code></td></tr>
<tr><td>Set manual negotiation.</td><td><code>SPI_Ethernet_24j600_MANUAL_NEGOTIATION</code></td></tr>
<tr><td>Set Half duplex Mode</td><td><code>SPI_Ethernet_24j600_HALFDUPLEX</code></td></tr>
<tr><td>Set Full duplex Mode</td><td><code>SPI_Ethernet_24j600_FULLDUPLEX</code></td></tr>
<tr><td>Set transmission speed of 10Mbps</td><td><code>SPI_Ethernet_24j600_SPD10</code></td></tr>
<tr><td>Set transmission speed of 100Mbps</td><td><code>SPI_Ethernet_24j600_SPD100</code></td></tr>
</table>

Note:

- It is advisable to use only the Auto-negotiation setting. If manual negotiation is used, then duplex and speed mode setting must be set also.
- Duplex and speed mode may be set only when using manual negotiation.</td></tr>
</table>

Returns	Nothing.
Requires	Global variables: - `SPI_Ethernet_24j600_CS`: Chip Select line - `SPI_Ethernet_24j600_CS_Direction`: Direction of the Chip Select pin - `SPI_Ethernet_24j600_RST`: Reset line - `SPI_Ethernet_24j600_RST_Direction`: Direction of the Reset pin must be defined before using this function. The SPI module needs to be initialized. See the SPIx_Init and SPIx_Init_Advanced routines.
Example	(see code below)
Notes	None.

```
// SPI Ethernet ENC24J600 module connections
var SPI_Ethernet_24j600_CS : sbit at RF1_bit;
var SPI_Ethernet_24j600_CS_Direction : sbit at TRISF1_bit;

var
  myMacAddr : array[6] of byte; // my MAC address
  myIpAddr  : array[4] of byte; // my IP addr
  ...
  myMacAddr[0] := 0x00;
  myMacAddr[1] := 0x14;
  myMacAddr[2] := 0xA5;
  myMacAddr[3] := 0x76;
  myMacAddr[4] := 0x19;
  myMacAddr[5] := 0x3F;

  myIpAddr[0]  := 192;
  myIpAddr[1]  := 168;
  myIpAddr[2]  := 1;
  myIpAddr[3]  := 60;

  SPI1_Init();
  SPI_Ethernet_24j600_Init(myMacAddr, myIpAddr, SPI_Ethernet_24j600_MANUAL_
NEGOTIATION and SPI_Ethernet_24j600_FULLDUPLEX and SPI_Ethernet_24j600_
SPD100);
```

SPI_Ethernet_24j600_Enable

Prototype

```
procedure SPI_Ethernet_24j600_Enable(enFlt : word);
```

Description

This is MAC module routine. This routine enables appropriate network traffic on the ENC24J600 module by the means of it's receive filters (unicast, multicast, broadcast, crc). Specific type of network traffic will be enabled if a corresponding bit of this routine's input parameter is set. Therefore, more than one type of network traffic can be enabled at the same time. For this purpose, predefined library constants (see the table below) can be ORed to form appropriate input value.

Advanced filtering available in the ENC24J600 module such as Pattern Match, Magic Packet and Hash Table can not be enabled by this routine. Additionaly, all filters, except CRC, enabled with this routine will work in OR mode, which means that packet will be received if any of the enabled filters accepts it.

This routine will change receive filter configuration on-the-fly. It will not, in any way, mess with enabling/disabling receive/transmit logic or any other part of the ENC24J600 module. The ENC24J600 module should be properly cofigured by the means of SPI_Ethernet_24j600_Init routine.

Parameters

- enFlt: network traffic/receive filter flags. Each bit corresponds to the appropriate network traffic/receive filter:

Bit	Mask	Description	Predefined library const
0	0x01	MAC Broadcast traffic/receive filter flag. When set, MAC broadcast traffic will be enabled.	_SPI_Ethernet_24j600_BROADCAST
1	0x02	MAC Multicast traffic/receive filter flag. When set, MAC multicast traffic will be enabled.	_SPI_Ethernet_24j600_MULTICAST
2	0x04	not used	none
3	0x08	not used	none
4	0x10	not used	none
5	0x20	CRC check flag. When set, packets with invalid CRC field will be discarded.	_SPI_Ethernet_24j600_CRC
6	0x40	not used	none
7	0x80	MAC Unicast traffic/receive filter flag. When set, MAC unicast traffic will be enabled.	_SPI_Ethernet_24j600_UNICAST

Returns

Nothing.

Requires

Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example

```
SPI_Ethernet_24j600_Enable(_SPI_Ethernet_24j600_CRC or _SPI_Ethernet_24j600_UNICAST); // enable CRC checking and Unicast traffic
```

Notes

Advanced filtering available in the ENC24J600 module such as Pattern Match, Magic Packet and Hash Table can not be enabled by this routine. Additionaly, all filters, except CRC, enabled with this routine will work in OR mode, which means that packet will be received if any of the enabled filters accepts it.

This routine will change receive filter configuration on-the-fly. It will not, in any way, mess with enabling/disabling receive/transmit logic or any other part of the ENC24J600 module. The ENC24J600 module should be properly cofigured by the means of SPI_Ethernet_24j600_Init routine.

SPI_Ethernet_24j600_Disable

Prototype	`procedure SPI_Ethernet_24j600_Disable(disFlt : word);`
Description	This is MAC module routine. This routine disables appropriate network traffic on the `ENC24J600` module by the means of it's receive filters (unicast, multicast, broadcast, crc). Specific type of network traffic will be disabled if a corresponding bit of this routine's input parameter is set. Therefore, more than one type of network traffic can be disabled at the same time. For this purpose, predefined library constants (see the table below) can be ORed to form appropriate input value.
Parameters	- `disFlt`: network traffic/receive filter flags. Each bit corresponds to the appropriate network traffic/receive filter: (see table below)
Returns	Nothing.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.
Example	`SPI_Ethernet_24j600_Disable(_SPI_Ethernet_24j600_CRC or _SPI_Ethernet_24j600_UNICAST); // disable CRC checking and Unicast traffic`
Notes	Advanced filtering available in the `ENC24J600` module such as `Pattern Match`, `Magic Packet` and `Hash Table` can not be disabled by this routine. This routine will change receive filter configuration on-the-fly. It will not, in any way, mess with enabling/disabling receive/transmit logic or any other part of the `ENC24J600` module. The `ENC24J600` module should be properly cofigured by the means of `SPI_Ethernet_24j600_Init` routine. The `ENC24J600` module should be properly cofigured by the means of SPI_Ethernet_24j600_Init routine.

Bit	Mask	Description	Predefined library const
0	`0x01`	MAC Broadcast traffic/receive filter flag. When set, MAC broadcast traffic will be disabled.	`_SPI_Ethernet_24j600_BROADCAST`
1	`0x02`	MAC Multicast traffic/receive filter flag. When set, MAC multicast traffic will be disabled.	`_SPI_Ethernet_24j600_MULTICAST`
2	`0x04`	not used	none
3	`0x08`	not used	none
4	`0x10`	not used	none
5	`0x20`	CRC check flag. When set, CRC check will be disabled and packets with invalid CRC field will be accepted.	`_SPI_Ethernet_24j600_CRC`
6	`0x40`	not used	none
7	`0x80`	MAC Unicast traffic/receive filter flag. When set, MAC unicast traffic will be disabled.	`_SPI_Ethernet_24j600_UNICAST`

SPI_Ethernet_24j600_doPacket

Prototype	`function SPI_Ethernet_24j600_doPacket() : byte;`
Description	This is MAC module routine. It processes next received packet if such exists. Packets are processed in the following manner: - ARP & ICMP requests are replied automatically. - upon TCP request the SPI_Ethernet_24j600_UserTCP function is called for further processing. - upon UDP request the SPI_Ethernet_24j600_UserUDP function is called for further processing.
Parameters	None.
Returns	- `0` - upon successful packet processing (zero packets received or received packet processed successfully). - `1` - upon reception error or receive buffer corruption. `ENC24J600` controller needs to be restarted. - `2` - received packet was not sent to us (not our IP, nor IP broadcast address). - `3` - received IP packet was not IPv4. - `4` - received packet was of type unknown to the library.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example	<pre>while true do
	  begin
	    ...
	    SPI_Ethernet_24j600_doPacket(); // process received packets
	    ...
	  end;</pre>

Notes	`SPI_Ethernet_24j600_doPacket` must be called as often as possible in user's code.

SPI_Ethernet_24j600_putByte

Prototype	`procedure SPI_Ethernet_24j600_putByte(v : byte);`
Description	This is MAC module routine. It stores one byte to address pointed by the current `ENC24J600` write pointer (`EWRPT`).
Parameters	- `v`: value to store
Returns	Nothing.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example	<pre>var
	  data : byte;
	  ...
	  SPI_Ethernet_24j600_putByte(data); // put an byte into ENC24J600 buffer</pre>

Notes	None.

SPI_Ethernet_24j600_putBytes

Prototype	`procedure SPI_Ethernet_24j600_putBytes(ptr : ^byte; n : word);`
Description	This is MAC module routine. It stores requested number of bytes into `ENC24J600` RAM starting from current `ENC24J600` write pointer (`EWRPT`) location.
Parameters	- `ptr`: RAM buffer containing bytes to be written into `ENC24J600` RAM. - `n`: number of bytes to be written.
Returns	Nothing.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example	`var`
	`  buffer : array[17] of byte;`
	`  ...`
	`  buffer := 'mikroElektronika';`
	`  ...`
	`  SPI_Ethernet_24j600_putBytes(buffer, 16); // put an RAM array into ENC24J600 buffer`

Notes	None.

SPI_Ethernet_24j600_putConstBytes

Prototype	`procedure SPI_Ethernet_24j600_putConstBytes(const ptr : ^byte; n : word);`
Description	This is MAC module routine. It stores requested number of const bytes into ENC24J600 RAM starting from current `ENC24J600` write pointer (`EWRPT`) location.
Parameters	- `ptr`: const buffer containing bytes to be written into `ENC24J600` RAM. - `n`: number of bytes to be written.
Returns	Nothing.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example	`const`
	`  buffer : array[17] of byte;`
	`  ...`
	`  buffer := 'mikroElektronika';`
	`  ...`
	`  SPI_Ethernet_24j600_putConstBytes(buffer, 16); // put a const array into ENC24J600 buffer`

Notes	None.

SPI_Ethernet_24j600_putString

Prototype	`function SPI_Ethernet_24j600_putString(ptr : ^byte) : word;`
Description	This is MAC module routine. It stores whole string (excluding null termination) into `ENC24J600` RAM starting from current `ENC24J600` write pointer (`EWRPT`) location.
Parameters	- `ptr`: string to be written into `ENC24J600` RAM.
Returns	Number of bytes written into `ENC24J600` RAM.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example	`var`
	`  buffer : string[16];`
	`  ...`
	`  buffer := 'mikroElektronika';`
	`  ...`
	`  SPI_Ethernet_24j600_putString(buffer); // put a RAM string into ENC24J600 buffer`

Notes	None.

SPI_Ethernet_24j600_putConstString

Prototype	`function SPI_Ethernet_24j600_putConstString(const ptr : ^byte) : word;`
Description	This is MAC module routine. It stores whole const string (excluding null termination) into `ENC24J600` RAM starting from current `ENC24J600` write pointer (`EWRPT`) location.
Parameters	- `ptr`: const string to be written into `ENC24J600` RAM.
Returns	Number of bytes written into `ENC24J600` RAM.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example	`const`
	`  buffer : string[16];`
	`  ...`
	`  buffer := 'mikroElektronika';`
	`  ...`
	`   SPI_Ethernet_24j600_putConstString(buffer); // put a const string into ENC24J600 buffer`

Notes	None.

SPI_Ethernet_24j600_getByte

Prototype	`function SPI_Ethernet_24j600_getByte() : byte;`
Description	This is MAC module routine. It fetches a byte from address pointed to by current `ENC24J600` read pointer (`ERDPT`).
Parameters	None.
Returns	Byte read from `ENC24J600` RAM.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example	`var`
	`  buffer : byte;`
	`  ...`
	`   buffer := SPI_Ethernet_24j600_getByte(); // read a byte from ENC24J600 buffer`

Notes	None.

SPI_Ethernet_24j600_getBytes

Prototype	`procedure SPI_Ethernet_24j600_getBytes(ptr : ^byte; addr : word; n : word);`
Description	This is MAC module routine. It fetches equested number of bytes from `ENC24J600` RAM starting from given address. If value of `0xFFFF` is passed as the address parameter, the reading will start from current `ENC24J600` read pointer (`ERDPT`) location.
Parameters	- `ptr`: buffer for storing bytes read from `ENC24J600` RAM. - `addr`: `ENC24J600` RAM start address. Valid values: `0..8192`. - `n`: number of bytes to be read.
Returns	Nothing.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example	`var`
	`  buffer: array[16] of byte;`
	`  ...`
	` SPI_Ethernet_24j600_getBytes(buffer, 0x100, 16); // read 16 bytes, starting from address 0x100`

Notes	None.

SPI_Ethernet_24j600_UserTCP

Prototype	`function SPI_Ethernet_24j600_UserTCP(var remoteHost : array[4] of byte; remotePort, localPort, reqLength : word; var flags: TEthj600PktFlags) : word;`
Description	This is TCP module routine. It is internally called by the library. The user accesses to the TCP request by using some of the SPI_Ethernet_24j600_get routines. The user puts data in the transmit buffer by using some of the SPI_Ethernet_24j600_put routines. The function must return the length in bytes of the TCP reply, or 0 if there is nothing to transmit. If there is no need to reply to the TCP requests, just define this function with return(0) as a single statement.

Parameters	- `remoteHost`: client's IP address.
	- `remotePort`: client's TCP port.
	- `localPort`: port to which the request is sent.
	- `reqLength`: TCP request data field length.
	- `flags`: structure consisted of two bit fields :
	
	Copy Code To Clipboard
	
	`type TEthj600PktFlags = record`
	`  canCloseTCP: boolean;  // flag which closes socket`
	`  isBroadcast: boolean;  // flag which denotes that the IP package has been received via subnet broadcast address`
	`end;`

Returns	- `0` - there should not be a reply to the request. - Length of TCP reply data field - otherwise.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.
Example	This function is internally called by the library and should not be called by the user's code.
Notes	The function source code is provided with appropriate example projects. The code should be adjusted by the user to achieve desired reply.

SPI_Ethernet_24j600_UserUDP

Prototype	`function SPI_Ethernet_24j600_UserUDP(var remoteHost : array[4] of byte; remotePort, destPort, reqLength : word; var flags: TEthj600PktFlags) : word;`
Description	This is UDP module routine. It is internally called by the library. The user accesses to the UDP request by using some of the SPI_Ethernet_24j600_get routines. The user puts data in the transmit buffer by using some of the SPI_Ethernet_24j600_put routines. The function must return the length in bytes of the UDP reply, or 0 if nothing to transmit. If you don't need to reply to the UDP requests, just define this function with a return(0) as single statement.

Parameters	- `remoteHost`: client's IP address.
	- `remotePort`: client's port.
	- `localPort`: port to which the request is sent.
	- `reqLength`: UDP request data field length.
	- `flags`: structure consisted of two bit fields :
	
	Copy Code To Clipboard
	
	`type TEthj600PktFlags = record`
	`  canCloseTCP: boolean;  // flag which closes socket (not relevant to UDP)`
	`  isBroadcast: boolean;  // flag which denotes that the IP package has been received via subnet broadcast address`
	`end;`

Returns	- `0` - there should not be a reply to the request. - Length of UDP reply data field - otherwise.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.
Example	This function is internally called by the library and should not be called by the user's code.
Notes	The function source code is provided with appropriate example projects. The code should be adjusted by the user to achieve desired reply.

SPI_Ethernet_24j600_getIpAddress

Prototype	`function SPI_Ethernet_24j600_getIpAddress() : word;`
Description	This routine should be used when DHCP server is present on the network to fetch assigned IP address.
Parameters	None.
Returns	Pointer to the global variable holding IP address.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example	`var`
	`  ipAddr : array[4] of byte;  // user IP address buffer`
	`  ...`
	`    memcpy(ipAddr, SPI_Ethernet_24j600_getIpAddress(), 4); // fetch IP address`

Notes	User should always copy the IP address from the RAM location returned by this routine into it's own IP address buffer. These locations should not be altered by the user in any case!

SPI_Ethernet_24j600_getGwIpAddress

Prototype	`function SPI_Ethernet_24j600_getGwIpAddress() : word;`
Description	This routine should be used when DHCP server is present on the network to fetch assigned gateway IP address.
Parameters	None.
Returns	Pointer to the global variable holding gateway IP address.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example	`var`
	`  gwIpAddr : array[4] of byte;  // user gateway IP address buffer`
	`  ...`
	`   memcpy(gwIpAddr, SPI_Ethernet_24j600_getGwIpAddress(), 4); // fetch gateway IP address`

Notes	User should always copy the IP address from the RAM location returned by this routine into it's own gateway IP address buffer. These locations should not be altered by the user in any case!

SPI_Ethernet_24j600_getDnsIpAddress

Prototype	`function SPI_Ethernet_24j600_getDnsIpAddress() : word;`
Description	This routine should be used when DHCP server is present on the network to fetch assigned DNS IP address.
Parameters	None.
Returns	Pointer to the global variable holding DNS IP address.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example	`var`
	`  dnsIpAddr : array[4] of byte;  // user DNS IP address buffer`
	`  ...`
	`  memcpy(dnsIpAddr, SPI_Ethernet_24j600_getDnsIpAddress(), 4); // fetch DNS server address`

Notes	User should always copy the IP address from the RAM location returned by this routine into it's own DNS IP address buffer. These locations should not be altered by the user in any case!

SPI_Ethernet_24j600_getIpMask

Prototype	`function SPI_Ethernet_24j600_getIpMask() : word;`
Description	This routine should be used when DHCP server is present on the network to fetch assigned DNS IP address.
Parameters	None.
Returns	Pointer to the global variable holding IP subnet mask.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example	<pre>var
	  IpMask : array[4] of byte; // user IP subnet mask buffer
	  ...
	   memcpy(IpMask, SPI_Ethernet_24j600_getIpMask(), 4); // fetch IP subnet mask</pre>

Notes	User should always copy the IP address from the RAM location returned by this routine into it's own IP subnet mask buffer. These locations should not be altered by the user in any case!

SPI_Ethernet_24j600_confNetwork

Prototype	`procedure SPI_Ethernet_24j600_confNetwork(var ipMask, gwIpAddr, dnsIpAddr : array[4] of byte);`
Description	Configures network parameters (IP subnet mask, gateway IP address, DNS IP address) when DHCP is not used.
Parameters	- `ipMask`: IP subnet mask. - `gwIpAddr` gateway IP address. - `dnsIpAddr`: DNS IP address.
Returns	Nothing.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example	<pre>var
	   ipMask : array[4] of byte; // network mask (for example : 255.255.255.0)
	  gwIpAddr : array[4] of byte; // gateway (router) IP address
	  dnsIpAddr : array[4] of byte; // DNS server IP address
	  ...
	  gwIpAddr[0] := 192;
	  gwIpAddr[1] := 168;
	  gwIpAddr[2] := 20;
	  gwIpAddr[3] := 6;
	
	  dnsIpAddr[0] := 192;
	  dnsIpAddr[1] := 168;
	  dnsIpAddr[2] := 20;
	  dnsIpAddr[3] := 100;
	
	  ipMask[0] := 255;
	  ipMask[1] := 255;
	  ipMask[2] := 255;
	  ipMask[3] := 0;
	  ...
	   SPI_Ethernet_24j600_confNetwork(ipMask, gwIpAddr, dnsIpAddr); // set network configuration parameters</pre>

Notes	The above mentioned network parameters should be set by this routine only if DHCP module is not used. Otherwise DHCP will override these settings.

SPI_Ethernet_24j600_arpResolve

Prototype	`function SPI_Ethernet_24j600_arpResolve(var ip : array[4] of byte; tmax : byte) : word;`
Description	This is ARP module routine. It sends an ARP request for given IP address and waits for ARP reply. If the requested IP address was resolved, an ARP cash entry is used for storing the configuration. ARP cash can store up to 3 entries.
Parameters	- `ip`: IP address to be resolved. - `tmax`: time in seconds to wait for an reply.
Returns	- MAC address behind the IP address - the requested IP address was resolved. - `0` - otherwise.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example	`var`
	`  IpAddr : array[4] of byte;  // IP address`
	`  ...`
	`  IpAddr[0] := 192;`
	`  IpAddr[0] := 168;`
	`  IpAddr[0] := 1;`
	`  IpAddr[0] := 1;`
	`  ...`
	`  SPI_Ethernet_24j600_arpResolve(IpAddr, 5); // get MAC address behind the above IP address, wait 5 secs for the response`

Notes	The Ethernet services are not stopped while this routine waits for ARP reply. The incoming packets will be processed normaly during this time.

SPI_Ethernet_24j600_sendUDP

Prototype	`function SPI_Ethernet_24j600_sendUDP(var destIP : array[4] of byte; sourcePort, destPort : word; pkt : ^byte; pktLen : word) : byte;`
Description	This is UDP module routine. It sends an UDP packet on the network.
Parameters	- `destIP`: remote host IP address. - `sourcePort`: local UDP source port number. - `destPort`: destination UDP port number. - `pkt`: packet to transmit. - `pktLen`: length in bytes of packet to transmit.
Returns	- `1` - UDP packet was sent successfully. - `0` - otherwise.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example	`var`
	`  IpAddr : array[4] of byte;  // remote IP address`
	`  ...`
	`  IpAddr[0] := 192;`
	`  IpAddr[0] := 168;`
	`  IpAddr[0] := 1;`
	`  IpAddr[0] := 1;`
	`  ...`
	`  SPI_Ethernet_24j600_sendUDP(IpAddr, 10001, 10001, 'Hello', 5); // send Hello message to the above IP address, from UDP port 10001 to UDP port 10001`

Notes	None.

SPI_Ethernet_24j600_dnsResolve

Prototype	`function SPI_Ethernet_24j600_dnsResolve(var host : string; tmax : byte) : word;`
Description	This is DNS module routine. It sends an DNS request for given host name and waits for DNS reply. If the requested host name was resolved, it's IP address is stored in library global variable and a pointer containing this address is returned by the routine. UDP port 53 is used as DNS port.
Parameters	- `host`: host name to be resolved. - `tmax`: time in seconds to wait for an reply.
Returns	- pointer to the location holding the IP address - the requested host name was resolved. - `0` - otherwise.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example	`var`
	`  remoteHostIpAddr : array[4] of byte;  // user host IP address buffer`
	`  ...`
	`  // SNTP server:`
	`   //  Zurich,  Switzerland:  Integrated  Systems  Lab,  Swiss  Fed.  Inst.  of Technology`
	`  // 129.132.2.21: swisstime.ethz.ch`
	`  // Service Area: Switzerland and Europe`
	`  memcpy(remoteHostIpAddr, SPI_Ethernet_24j600_dnsResolve('swisstime.ethz.ch', 5), 4);`

Notes	The Ethernet services are not stopped while this routine waits for DNS reply. The incoming packets will be processed normaly during this time. User should always copy the IP address from the RAM location returned by this routine into it's own resolved host IP address buffer. These locations should not be altered by the user in any case!

SPI_Ethernet_24j600_initDHCP

Prototype	`function SPI_Ethernet_24j600_initDHCP(tmax : byte) : byte;`
Description	This is DHCP module routine. It sends an DHCP request for network parameters (IP, gateway, DNS addresses and IP subnet mask) and waits for DHCP reply. If the requested parameters were obtained successfully, their values are stored into the library global variables. These parameters can be fetched by using appropriate library IP get routines: - SPI_Ethernet_24j600_getIpAddress - fetch IP address. - SPI_Ethernet_24j600_getGwIpAddress - fetch gateway IP address. - SPI_Ethernet_24j600_getDnsIpAddress - fetch DNS IP address. - SPI_Ethernet_24j600_getIpMask - fetch IP subnet mask. UDP port `68` is used as DHCP client port and UDP port `67` is used as DHCP server port.
Parameters	- `tmax`: time in seconds to wait for an reply.
Returns	- `1` - network parameters were obtained successfully. - `0` - otherwise.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example	`...`
	`SPI_Ethernet_24j600_initDHCP(5); // get network configuration from DHCP server, wait 5 sec for the response`
	`...`

Notes	The Ethernet services are not stopped while this routine waits for DNS reply. The incoming packets will be processed normaly during this time. When DHCP module is used, global library variable `SPI_Ethernet_24j600_userTimerSec` is used to keep track of time. It is user responsibility to increment this variable each second in it's code.

SPI_Ethernet_24j600_doDHCPLeaseTime

Prototype	`function SPI_Ethernet_24j600_doDHCPLeaseTime() : byte;`
Description	This is DHCP module routine. It takes care of IP address lease time by decrementing the global lease time library counter. When this time expires, it's time to contact DHCP server and renew the lease.
Parameters	None.
Returns	- `0` - lease time has not expired yet. - `1` - lease time has expired, it's time to renew it.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example	<pre>while true do
	  begin
	    ...
	    if (SPI_Ethernet_24j600_doDHCPLeaseTime() <> 0) then
	      begin
	        ... // it's time to renew the IP address lease
	      end;
	  end;</pre>

Notes	None.

SPI_Ethernet_24j600_renewDHCP

Prototype	`function SPI_Ethernet_24j600_renewDHCP(tmax : byte) : byte;`
Description	This is DHCP module routine. It sends IP address lease time renewal request to DHCP server.
Parameters	- `tmax`: time in seconds to wait for an reply.
Returns	- `1` - upon success (lease time was renewed). - `0` - otherwise (renewal request timed out).
Requires	Ethernet module has to be initialized. See SPI_Ethernet_24j600_Init.

Example	<pre>while true do
	  begin
	    ...
	    if (SPI_Ethernet_24j600_doDHCPLeaseTime() <> 0) then
	      begin
	          SPI_Ethernet_24j600_renewDHCP(5); // it's time to renew the IP
	address lease, with 5 secs for a reply
	      end;
	    ...
	  end;</pre>

Notes	None.

SPI Graphic Lcd Library

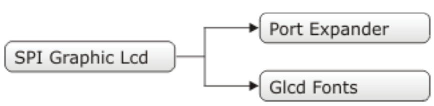
mikroPascal PRO for PIC32 provides a library for operating Graphic Lcd 128x64 (with commonly used Samsung KS108/KS107 controller) via SPI interface.

For creating a custom set of Glcd images use Glcd Bitmap Editor Tool.

Important:

- When using this library with PIC32 family MCUs be aware of their voltage incompatibility with certain number of Samsung KS0108 based Glcd modules.
  So, additional external power supply for these modules may be required.
- Library uses the SPI module for communication. The user must initialize the appropriate SPI module before using the SPI Glcd Library.
- For MCUs with multiple SPI modules it is possible to initialize all of them and then switch by using the `SPI_Set_Active()` routine. See the SPI Library functions.
- This Library is designed to work with the mikroElektronika's Serial Lcd/Glcd Adapter Board pinout, see schematic at the bottom of this page for details..

Library Dependency Tree



External dependencies of SPI Lcd Library

The implementation of SPI Lcd Library routines is based on Port Expander Library routines.

External dependencies are the same as Port Expander Library external dependencies.

Library Routines

Basic routines:

- SPI_Glcd_Init
- SPI_Glcd_Set_Side
- SPI_Glcd_Set_Page
- SPI_Glcd_Set_X
- SPI_Glcd_Read_Data
- SPI_Glcd_Write_Data

Advanced routines:

- SPI_Glcd_Fill
- SPI_Glcd_Dot
- SPI_Glcd_Line
- SPI_Glcd_V_Line
- SPI_Glcd_H_Line

- SPI_Glcd_Rectangle
- SPI_Glcd_Rectangle_Round_Edges
- SPI_Glcd_Rectangle_Round_Edges_Fill
- SPI_Glcd_Box
- SPI_Glcd_Circle
- SPI_Glcd_Circle_Fill
- SPI_Glcd_Set_Font
- SPI_Glcd_Write_Char
- SPI_Glcd_Write_Text
- SPI_Glcd_Image
- SPI_Glcd_PartialImage

SPI_Glcd_Init

Prototype	`procedure SPI_Glcd_Init(DeviceAddress : byte);`
Description	Initializes the Glcd module via SPI interface.
Parameters	- `DeviceAddress`: SPI expander hardware address, see schematic at the bottom of this page
Returns	Nothing.
Requires	Global variables: - `SPExpanderCS`: Chip Select line - `SPExpanderRST`: Reset line - `SPExpanderCS_Direction`: Direction of the Chip Select pin - `SPExpanderRST_Direction`: Direction of the Reset pin must be defined before using this function. The SPI module needs to be initialized. See SPIx_Init and SPIx_Init_Advanced routines.

Example	`// Port Expander module connections`
	`var SPExpanderRST : sbit at LATF0_bit;`
	`    SPExpanderCS  : sbit at LATF1_bit;`
	`    SPExpanderRST_Direction : sbit at TRISF0_bit;`
	`    SPExpanderCS_Direction  : sbit at TRISF1_bit;`
	`// End Port Expander module connections`
	`...`
	
	`// If Port Expander Library uses SPI module :`
	`SPI1_Init();  // Initialize SPI module used with PortExpander`
	`SPI_Glcd_Init(0);`

Notes	None.

SPI_Glcd_Set_Side

Prototype	`procedure SPI_Glcd_Set_Side(x_pos : byte);`
Description	Selects Glcd side. Refer to the Glcd datasheet for detail explanation.
Parameters	- `x_pos`: position on x-axis. Valid values: 0..127 The parameter `x_pos` specifies the Glcd side: values from 0 to 63 specify the left side, values from 64 to 127 specify the right side.
Returns	Nothing.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine.

Example	The following two lines are equivalent, and both of them select the left side of Glcd:
	
	`SPI_Glcd_Set_Side(0);`
	`SPI_Glcd_Set_Side(10);`

Notes	For side, x axis and page layout explanation see schematic at the bottom of this page.

SPI_Glcd_Set_Page

Prototype	`procedure SPI_Glcd_Set_Page(page : byte);`
Description	Selects page of Glcd.
Returns	- `page`: page number. Valid values: 0..7
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine.
Example	`SPI_Glcd_Set_Page(5);`
Notes	For side, x axis and page layout explanation see schematic at the bottom of this page.

SPI_Glcd_Set_X

Prototype	`procedure SPI_Glcd_Set_X(x_pos : byte);`
Description	Sets x-axis position to `x_pos` dots from the left border of Glcd within the selected side.
Parameters	- `x_pos`: position on x-axis. Valid values: 0..63
Returns	Nothing.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine.
Example	`SPI_Glcd_Set_X(25);`
Notes	For side, x axis and page layout explanation see schematic at the bottom of this page.

SPI_Glcd_Read_Data

Prototype	`function SPI_Glcd_Read_Data() : byte;`
Description	Reads data from the current location of Glcd memory and moves to the next location.
Returns	One byte from Glcd memory.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine. Glcd side, x-axis position and page should be set first. See the functions SPI_Glcd_Set_Side, SPI_Glcd_Set_X, and SPI_Glcd_Set_Page.
Parameters	None.

Example	`var data_ : byte;`
	`...`
	`data_ := SPI_Glcd_Read_Data();`

Notes	None.

SPI_Glcd_Write_Data

Prototype	`procedure SPI_Glcd_Write_Data(data_ : byte);`
Description	Writes one byte to the current location in Glcd memory and moves to the next location.
Parameters	- `data_`: data to be written
Returns	Nothing.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine. Glcd side, x-axis position and page should be set first. See the functions SPI_Glcd_Set_Side, SPI_Glcd_Set_X, and SPI_Glcd_Set_Page.

Example	`var data_ : byte;`
	`...`
	`SPI_Glcd_Write_Data(data_);`

Notes	None.

SPI_Glcd_Fill

Prototype	`procedure SPI_Glcd_Fill(pattern : byte);`
Description	Fills Glcd memory with byte pattern. To clear the Glcd screen, use `SPI_Glcd_Fill(0)`. To fill the screen completely, use `SPI_Glcd_Fill(0xFF)`.
Parameters	- `pattern`: byte to fill Glcd memory with
Returns	Nothing.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine.
Example	`// Clear screen` `SPI_Glcd_Fill(0);`
Notes	None.

SPI_Glcd_Dot

Prototype	`procedure SPI_Glcd_Dot(x_pos, y_pos, color : byte);`
Description	Draws a dot on Glcd at coordinates (`x_pos, y_pos`).
Parameters	- `x_pos`: x position. Valid values: 0..127 - `y_pos`: y position. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the dot state: 0 clears dot, 1 puts a dot, and 2 inverts dot state.
Returns	Nothing.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine.
Example	`// Invert the dot in the upper left corner` `SPI_Glcd_Dot(0, 0, 2);`
Notes	For x and y axis layout explanation see schematic at the bottom of this page.

SPI_Glcd_Line

Prototype	`procedure SPI_Glcd_Line(x_start, y_start, x_end, y_end : integer; color : byte);`
Description	Draws a line on Glcd. Parameters:
Parameters	- `x_start`: x coordinate of the line start. Valid values: 0..127 - `y_start`: y coordinate of the line start. Valid values: 0..63 - `x_end`: x coordinate of the line end. Valid values: 0..127 - `y_end`: y coordinate of the line end. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 Parameter `color` determines the line color: 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine.
Example	`// Draw a line between dots (0,0) and (20,30)` `SPI_Glcd_Line(0, 0, 20, 30, 1);`
Notes	None.

SPI_Glcd_V_Line

Prototype	`procedure SPI_Glcd_V_Line(y_start, y_end, x_pos, color : byte);`
Description	Draws a vertical line on Glcd.
Parameters	- `y_start`: y coordinate of the line start. Valid values: 0..63 - `y_end`: y coordinate of the line end. Valid values: 0..63 - `x_pos`: x coordinate of vertical line. Valid values: 0..127 - `color`: color parameter. Valid values: 0..2 Parameter `color` determines the line color: 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine.
Example	`// Draw a vertical line between dots (10,5) and (10,25)` `SPI_Glcd_V_Line(5, 25, 10, 1);`
Notes	None.

SPI_Glcd_H_Line

Prototype	`procedure SPI_Glcd_H_Line(x_start, x_end, y_pos, color : byte);`
Description	Draws a horizontal line on Glcd.
Parameters	- `x_start`: x coordinate of the line start. Valid values: 0..127 - `x_end`: x coordinate of the line end. Valid values: 0..127 - `y_pos`: y coordinate of horizontal line. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the line color: 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine.
Example	`// Draw a horizontal line between dots (10,20) and (50,20)` `SPI_Glcd_H_Line(10, 50, 20, 1);`
Notes	None.

SPI_Glcd_Rectangle

Prototype	`procedure SPI_Glcd_Rectangle(x_upper_left, y_upper_left, x_bottom_right, y_bottom_right, color : byte);`
Description	Draws a rectangle on Glcd.
Parameters	- `x_upper_left`: x coordinate of the upper left rectangle corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left rectangle corner. Valid values: 0..63 - `x_bottom_right`: x coordinate of the lower right rectangle corner. Valid values: 0..127 - `y_bottom_right`: y coordinate of the lower right rectangle corner. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the color of the rectangle border: 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine.
Example	`// Draw a rectangle between dots (5,5) and (40,40)` `SPI_Glcd_Rectangle(5, 5, 40, 40, 1);`
Notes	None.

SPI_Glcd_Rectangle_Round_Edges

Prototype	`procedure SPI_Glcd_Rectangle_Round_Edges(x_upper_left : byte; y_upper_left : byte; x_bottom_right : byte; y_bottom_right : byte; radius : byte; color : byte);`
Description	Draws a rounded edge rectangle on Glcd.
Parameters	- `x_upper_left`: x coordinate of the upper left rectangle corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left rectangle corner. Valid values: 0..63 - `x_bottom_right`: x coordinate of the lower right rectangle corner. Valid values: 0..127 - `y_bottom_right`: y coordinate of the lower right rectangle corner. Valid values: 0..63 - `round_radius`: radius of the rounded edge. - `color`: color parameter. Valid values: 0..2
Returns	Nothing.
Requires	Glcd needs to be initialized, see SPI_Glcd_Init routine.
Example	`// Draws a rounded edge rectangle between dots (5,5) and (40,40) with radius` `SPI_Glcd_Rectangle_Round_Edges(5, 5, 40, 40, 12, 1);`
Notes	None.

SPI_Glcd_Rectangle_Round_Edges_Fill

Prototype	`procedure SPI_Glcd_Rectangle_Round_Edges_Fill(x_upper_left : byte; y_upper_left : byte; x_bottom_right : byte; y_bottom_right : byte; radius : byte; color : byte);`
Description	Draws a filled rounded edge rectangle on Glcd with color.
Parameters	- `x_upper_left`: x coordinate of the upper left rectangle corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left rectangle corner. Valid values: 0..63 - `x_bottom_right`: x coordinate of the lower right rectangle corner. Valid values: 0..127 - `y_bottom_right`: y coordinate of the lower right rectangle corner. Valid values: 0..63 - `round_radius`: radius of the rounded edge - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the color of the rectangle border: 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized, see SPI_Glcd_Init routine.
Example	`// Draws a filled rounded edge rectangle between dots (5,5) and (40,40) with the edge radius of 12` `SPI_Glcd_Rectangle_Round_Edges(5, 5, 40, 40, 12, 1);`
Notes	None.

SPI_Glcd_Box

Prototype	`procedure SPI_Glcd_Box(x_upper_left, y_upper_left, x_bottom_right, y_bottom_right, color : byte);`
Description	Draws a box on Glcd.
Parameters	- `x_upper_left`: x coordinate of the upper left box corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left box corner. Valid values: 0..63 - `x_bottom_right`: x coordinate of the lower right box corner. Valid values: 0..127 - `y_bottom_right`: y coordinate of the lower right box corner. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the color of the box fill: 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine.
Example	`// Draw a box between dots (5,15) and (20,40)` `SPI_Glcd_Box(5, 15, 20, 40, 1);`
Notes	None.

SPI_Glcd_Circle

Prototype	`procedure SPI_Glcd_Circle(x_center, y_center, radius : integer; color : byte);`
Description	Draws a circle on Glcd.
Parameters	- `x_center`: x coordinate of the circle center. Valid values: 0..127 - `y_center`: y coordinate of the circle center. Valid values: 0..63 - `radius`: radius size - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the color of the circle line: 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine.
Example	`// Draw a circle with center in (50,50) and radius=10` `SPI_Glcd_Circle(50, 50, 10, 1);`
Notes	None.

SPI_Glcd_Circle_FIll

Prototype	`procedure SPI_Glcd_Circle_Fill(x_center : integer; y_center : integer; radius : integer; color : byte);`
Description	Draws a filled circle on Glcd.
Parameters	- `x_center`: x coordinate of the circle center. Valid values: 0..127 - `y_center`: y coordinate of the circle center. Valid values: 0..63 - `radius`: radius size - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the color of the circle : 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine.
Example	`// Draw a circle with center in (50,50) and radius=10` `SPI_Glcd_Circle_Fill(50, 50, 10, 1);`
Notes	None.

SPI_Glcd_Set_Font

Prototype	`procedure SPI_Glcd_Set_Font(activeFont: LongInt; aFontWidth, aFontHeight : byte; aFontOffs : word);`
Description	Sets font that will be used with SPI_Glcd_Write_Char and SPI_Glcd_Write_Text routines.
Parameters	None.
Returns	- `activeFont`: font to be set. Needs to be formatted as an array of char - `aFontWidth`: width of the font characters in dots. - `aFontHeight`: height of the font characters in dots. - `aFontOffs`: number that represents difference between the mikroPascal PRO for PIC32 character set and regular ASCII set (eg. if 'A' is 65 in ASCII character, and 'A' is 45 in the mikroPascal PRO for PIC32 character set, aFontOffs is 20). Demo fonts supplied with the library have an offset of 32, which means that they start with space. The user can use fonts given in the file `__Lib_GLCDFonts` file located in the Uses folder or create his own fonts. List of supported fonts: - `Font_Glcd_System3x5` - `Font_Glcd_System5x7` - `Font_Glcd_5x7` - `Font_Glcd_Character8x7` For the sake of the backward compatibility, these fonts are supported also: - `System3x5` (equivalent to `Font_Glcd_System3x5`) - `FontSystem5x7_v2` (equivalent to `Font_Glcd_System5x7`) - `font5x7` (equivalent to `Font_Glcd_5x7`) - `Character8x7` (equivalent to `Font_Glcd_Character8x7`)
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine.
Example	`// Use the custom 5x7 font "myfont" which starts with space (32):` `SPI_Glcd_Set_Font(myfont, 5, 7, 32);`
Notes	None.

SPI_Glcd_Write_Char

Prototype	`procedure SPI_Glcd_Write_Char(chr1, x_pos, page_num, color : byte);`
Description	Prints character on Glcd.
Parameters	- `chr1`: character to be written - `x_pos`: character starting position on x-axis. Valid values: 0..(127-FontWidth) - `page_num`: the number of the page on which character will be written. Valid values: 0..7 - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the color of the character: 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine. Use the SPI_Glcd_Set_Font to specify the font for display; if no font is specified, then the default `Font_Glcd_System5x7` font supplied with the library will be used.
Example	`// Write character 'C' on the position 10 inside the page 2:` `SPI_Glcd_Write_Char('C', 10, 2, 1);`
Notes	For x axis and page layout explanation see schematic at the bottom of this page.

SPI_Glcd_Write_Text

Prototype	`procedure SPI_Glcd_Write_Text(var text: array[40] of char; x_pos, page_num, color : byte);`
Description	Prints text on Glcd.
Parameters	- `text`: text to be written - `x_pos`: text starting position on x-axis. - `page_num`: the number of the page on which text will be written. Valid values: 0..7 - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the color of the text: 0 white, 1 black, and 2 inverts each dot.
Returns	Nothing.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine. Use the SPI_Glcd_Set_Font to specify the font for display; if no font is specified, then the default `Font_Glcd_System5x7` font supplied with the library will be used.
Example	`// Write text "Hello world!" on the position 10 inside the page 2:` `SPI_Glcd_Write_Text("Hello world!", 10, 2, 1);`
Notes	For x axis and page layout explanation see schematic at the bottom of this page.

SPI_Glcd_Image

Prototype	`procedure SPI_Glcd_Image(const image: ^byte);`
Description	Displays bitmap on Glcd.
Parameters	- `image`: image to be displayed. Bitmap array can be located in both code and RAM memory (due to the mikroPascal PRO for PIC32 pointer to const and pointer to RAM equivalency).
Returns	Nothing.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine.
Example	`// Draw image my_image on Glcd` `SPI_Glcd_Image(my_image);`
Notes	Use the mikroPascal PRO for PIC32 integrated Glcd Bitmap Editor, **Tools > Glcd Bitmap Editor**, to convert image to a constant array suitable for displaying on Glcd.

SPI_Glcd_PartialImage

Prototype	`procedure SPI_Glcd_PartialImage(x_left, y_top, width, height, picture_width, picture_height : word; const image : ^byte);`
Description	Displays a partial area of the image on a desired location.
Parameters	- `x_left`: x coordinate of the desired location (upper left coordinate). - `y_top`: y coordinate of the desired location (upper left coordinate). - `width`: desired image width. - `height`: desired image height. - `picture_width`: width of the original image. - `picture_height`: height of the original image. - `image`: image to be displayed. Bitmap array can be located in both code and RAM memory (due to the mikroPascal PRO for PIC pointer to const and pointer to RAM equivalency).
Returns	Nothing.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine.
Example	`// Draws a 10x15 part of the image starting from the upper left corner on the coordinate (10,12). Original image size is 16x32.` `SPI_Glcd_PartialImage(10, 12, 10, 15, 16, 32, @image);`
Notes	Use the mikroPascal PRO for PIC32 integrated Glcd Bitmap Editor, Tools > Glcd Bitmap Editor, to convert image to a constant array suitable for displaying on Glcd.

Library Example

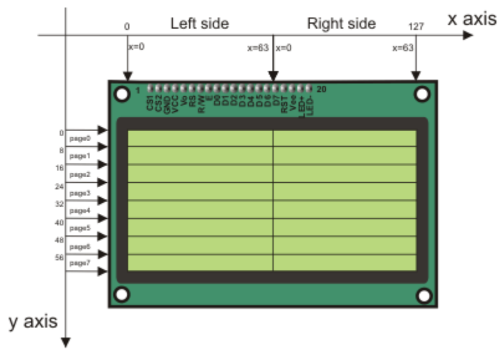
The example demonstrates how to communicate to KS0108 Glcd via the SPI module, using serial to parallel convertor MCP23S17.

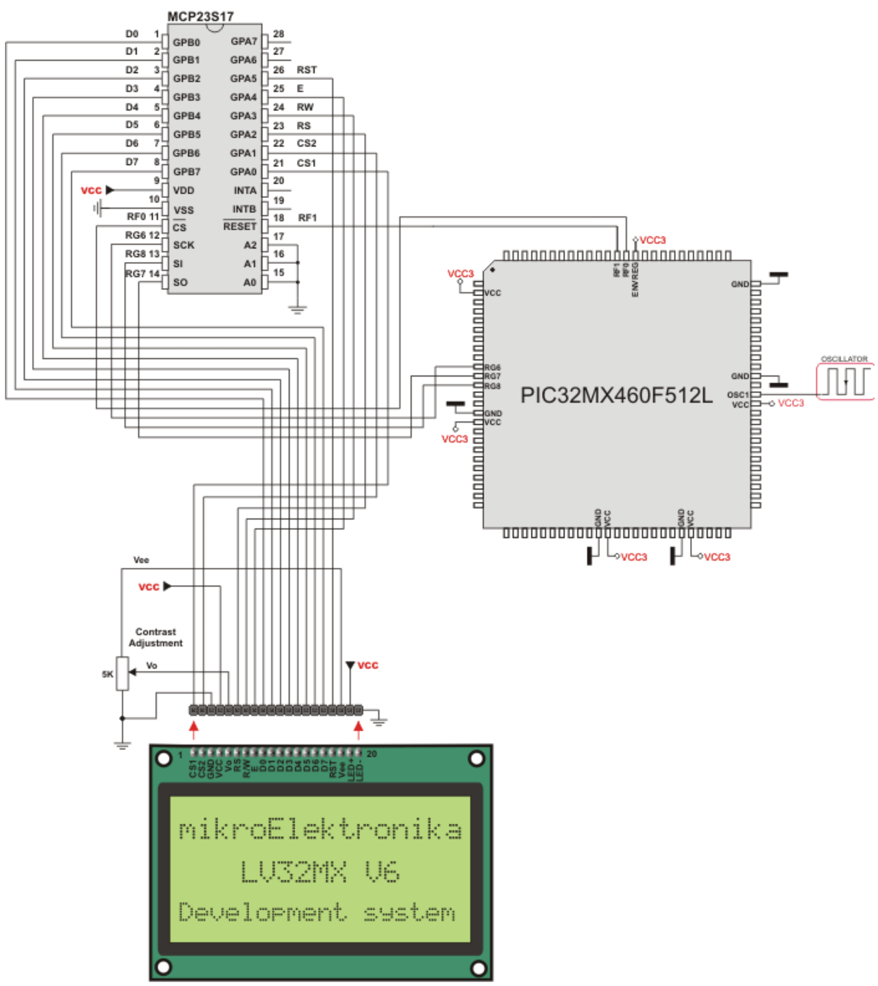
Copy Code To Clipboard

```
program SPI_Glcd;

// Port Expander module connections
var SPExpanderRST : sbit at LATD8_bit;
    SPExpanderCS  : sbit at LATD9_bit;
    SPExpanderRST_Direction : sbit at TRISD8_bit;
    SPExpanderCS_Direction  : sbit at TRISD9_bit;
// End Port Expander module connections

var someText : array[20] of char;
    counter : byte;

procedure Delay2S;
  begin
    Delay_ms(2000);
  end;

begin
  CHECON := 0x32;
  AD1PCFG := 0xFFFF;                          // Configure AN pins as digital

  // If Port Expander Library uses SPI2 module
  // Initialize SPI module used with PortExpander
     SPI2_Init_Advanced(_SPI_MASTER,_SPI_8_BIT,   4,   _SPI_SS_DISABLE,_SPI_DATA_SAMPLE_
MIDDLE,_SPI_CLK_IDLE_LOW,_SPI_ACTIVE_2_IDLE);

  SPI_Glcd_Init(0);                                     // Initialize Glcd via SPI
  SPI_Glcd_Fill(0x00);                                  // Clear Glcd

  while TRUE do
    begin
      SPI_Glcd_Image(@truck_bmp);                       // Draw image
      Delay2s(); Delay2s();
      SPI_Glcd_fill(0x00);                              // Clear GLCD
      SPI_Glcd_PartialImage(0,0,68,30,128,64,@truck_bmp);  // Partial image
      Delay_ms(500);
      SPI_Glcd_PartialImage(24,16,68,30,128,64,@truck_bmp);
      Delay_ms(500);
      SPI_Glcd_PartialImage(56,34,68,30,128,64,@truck_bmp);
      Delay2s(); Delay2s();
      SPI_Glcd_Fill(0x00);                              // Clear GLCD

      SPI_Glcd_Box(62,40,124,56,1);                     // Draw box
      SPI_Glcd_Rectangle(5,5,84,35,1);                  // Draw rectangle
      Delay_ms(1000);
      SPI_Glcd_Rectangle_Round_Edges(2,2,87,38,7,1);
      Delay_ms(1000);
      SPI_Glcd_Rectangle_Round_Edges_Fill(8,8,81,32,12,1);
      Delay_ms(1000);
      SPI_Glcd_Line(0, 0, 127, 63, 1);                  // Draw line
      Delay2s();
```

```
      counter := 5;

      while (counter <= 59) do                   // Draw horizontal and vertical lines
        begin
          Delay_ms(250);
          SPI_Glcd_V_Line(2, 54, counter, 1);
          SPI_Glcd_H_Line(2, 120, counter, 1);
          Counter := counter + 5;
        end;

      Delay2S();

      SPI_Glcd_Fill(0x00);                                    // Clear GLCD

          SPI_Glcd_Set_Font(@Font_Glcd_Character8x7,  8,  7,  32);   //  Choose  font
"Character8x7"
      SPI_Glcd_Write_Text('mikroE', 1, 7, 2);                 // Write string

      for counter := 1 to 10 do                               // Draw circles
        SPI_Glcd_Circle(63,32, 3*counter, 1);
      Delay2S();

      SPI_Glcd_Circle_Fill(63,32, 30, 1);                     // Draw circles
      Delay2S();

      SPI_Glcd_Box(10,20, 70,63, 2);                          // Draw box
      Delay2s();

      SPI_Glcd_Fill(0xFF);                                    // Fill Glcd

      SPI_Glcd_Set_Font(@Font_Glcd_Character8x7, 8, 7, 32);   // Change font
      someText := '8x7 Font';
      SPI_Glcd_Write_Text(someText, 5, 0, 2);                 // Write string
      Delay2s();

      SPI_Glcd_Set_Font(@Font_Glcd_System3x5, 3, 5, 32);      // Change font
      someText := '3X5 CAPITALS ONLY';
      SPI_Glcd_Write_Text(someText, 60, 2, 2);                // Write string
      Delay2s();

      SPI_Glcd_Set_Font(@Font_Glcd_System5x7, 5, 7, 32);      // Change font
      someText := '5x7 Font';
      SPI_Glcd_Write_Text(someText, 5, 4, 2);                 // Write string
      Delay2s();

      SPI_Glcd_Set_Font(@Font_Glcd_5x7, 5, 7, 32);            // Change font
      someText := '5x7 Font (v2)';
      SPI_Glcd_Write_Text(someText, 50, 6, 2);                // Write string
      Delay2s();
    end;
end.
```

HW Connection





SPI Glcd HW connection

SPI Lcd Library

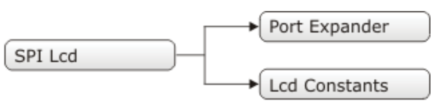
The mikroPascal PRO for PIC32 provides a library for communication with Lcd (with HD44780 compliant controllers) in 4-bit mode via SPI interface.

For creating a custom set of Lcd characters use Lcd Custom Character Tool.

Important:

- When using this library with PIC32 family MCUs be aware of their voltage incompatibility with certain number of Lcd modules.
  So, additional external power supply for these modules may be required.
- Library uses the SPI module for communication. The user must initialize the appropriate SPI module before using the SPI Lcd Library.
- For MCUs with multiple SPI modules it is possible to initialize all of them and then switch by using the SPI_Set_Active() routine. See the SPI Library functions.
- This Library is designed to work with the mikroElektronika's Serial Lcd Adapter Board pinout, see schematic at the bottom of this page for details.

Library Dependency Tree



External dependencies of SPI Lcd Library

The implementation of SPI Lcd Library routines is based on Port Expander Library routines.

External dependencies are the same as Port Expander Library external dependencies.

Library Routines

- SPI_Lcd_Config
- SPI_Lcd_Out
- SPI_Lcd_Out_Cp
- SPI_Lcd_Chr
- SPI_Lcd_Chr_Cp
- SPI_Lcd_Cmd

SPI_Lcd_Config

Prototype	`procedure SPI_Lcd_Config(DeviceAddress : byte);`
Description	Initializes the Lcd module via SPI interface.
Parameters	- `DeviceAddress`: SPI expander hardware address, see schematic at the bottom of this page
Returns	Nothing.
Requires	Global variables: - `SPExpanderCS`: Chip Select line - `SPExpanderRST`: Reset line - `SPExpanderCS_Direction`: Direction of the Chip Select pin - `SPExpanderRST_Direction`: Direction of the Reset pin must be defined before using this function. The SPI module needs to be initialized. See `SPIx_Init` and `SPIx_Init_Advanced` routines.

Example	`// Port Expander module connections`
	`var SPExpanderRST : sbit at LATF0_bit;`
	`var SPExpanderCS  : sbit at LATF1_bit;`
	`var SPExpanderRST_Direction : sbit at TRISF0_bit;`
	`var SPExpanderCS_Direction  : sbit at TRISF1_bit;`
	`// End Port Expander module connections`
	
	`// If Port Expander Library uses SPI1 module`
	`SPI1_Init();                              // Initialize SPI module used with PortExpander`
	`SPI_Lcd_Config(0);                        // initialize Lcd over SPI interface`

Notes	None.

SPI_Lcd_Out

Prototype	`procedure SPI_Lcd_Out(row, column : byte; var text : string);`
Description	Prints text on the Lcd starting from specified position. Both string variables and literals can be passed as a text.
Parameters	- `row`: starting position row number - `column`: starting position column number - `text`: text to be written
Returns	Nothing.
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd_Config routine.
Example	`// Write text "Hello!" on Lcd starting from row 1, column 3:` `SPI_Lcd_Out(1, 3, "Hello!");`
Notes	None.

SPI_Lcd_Out_Cp

Prototype	`procedure SPI_Lcd_Out_CP(var text : string); // write text at current pos`
Description	Prints text on the Lcd at current cursor position. Both string variables and literals can be passed as a text.
Parameters	- `text`: text to be written
Returns	Nothing.
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd_Config routine.
Example	`// Write text "Here!" at current cursor position:` `SPI_Lcd_Out_CP("Here!");`
Notes	None.

SPI_Lcd_Chr

Prototype	`procedure SPI_Lcd_Chr(Row, Column, Out_Char : byte);`
Description	Prints character on Lcd at specified position. Both variables and literals can be passed as character.
Parameters	- `Row`: writing position row number - `Column`: writing position column number - `Out_Char`: character to be written
Returns	Nothing.
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd_Config routine.
Example	`// Write character "i" at row 2, column 3:` `SPI_Lcd_Chr(2, 3, 'i');`
Notes	None.

SPI_Lcd_Chr_Cp

Prototype	`procedure SPI_Lcd_Chr_CP(Out_Char : byte);`
Description	Prints character on Lcd at current cursor position. Both variables and literals can be passed as character.
Parameters	- `Out_Char`: character to be written
Returns	Nothing.
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd_Config routine.
Example	`// Write character "e" at current cursor position:` `SPI_Lcd_Chr_Cp('e');`
Notes	None.

SPI_Lcd_Cmd

Prototype	`procedure SPI_Lcd_Cmd(out_char : byte);`
Description	Sends command to Lcd.
Parameters	- `out_char`: command to be sent
Returns	Nothing.
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd_Config routine.
Example	`// Clear Lcd display:` `SPI_Lcd_Cmd(_LCD_CLEAR);`
Notes	Predefined constants can be passed to the routine, see Available SPI Lcd Commands.

Available SPI Lcd Commands

SPI Lcd Command	Purpose
`_LCD_FIRST_ROW`	Move cursor to the 1st row
`_LCD_SECOND_ROW`	Move cursor to the 2nd row
`_LCD_THIRD_ROW`	Move cursor to the 3rd row
`_LCD_FOURTH_ROW`	Move cursor to the 4th row
`_LCD_CLEAR`	Clear display
`_LCD_RETURN_HOME`	Return cursor to home position, returns a shifted display to its original position. Display data RAM is unaffected.
`_LCD_CURSOR_OFF`	Turn off cursor
`_LCD_UNDERLINE_ON`	Underline cursor on
`_LCD_BLINK_CURSOR_ON`	Blink cursor on
`_LCD_MOVE_CURSOR_LEFT`	Move cursor left without changing display data RAM
`_LCD_MOVE_CURSOR_RIGHT`	Move cursor right without changing display data RAM
`_LCD_TURN_ON`	Turn Lcd display on
`_LCD_TURN_OFF`	Turn Lcd display off
`_LCD_SHIFT_LEFT`	Shift display left without changing display data RAM
`_LCD_SHIFT_RIGHT`	Shift display right without changing display data RAM

Library Example

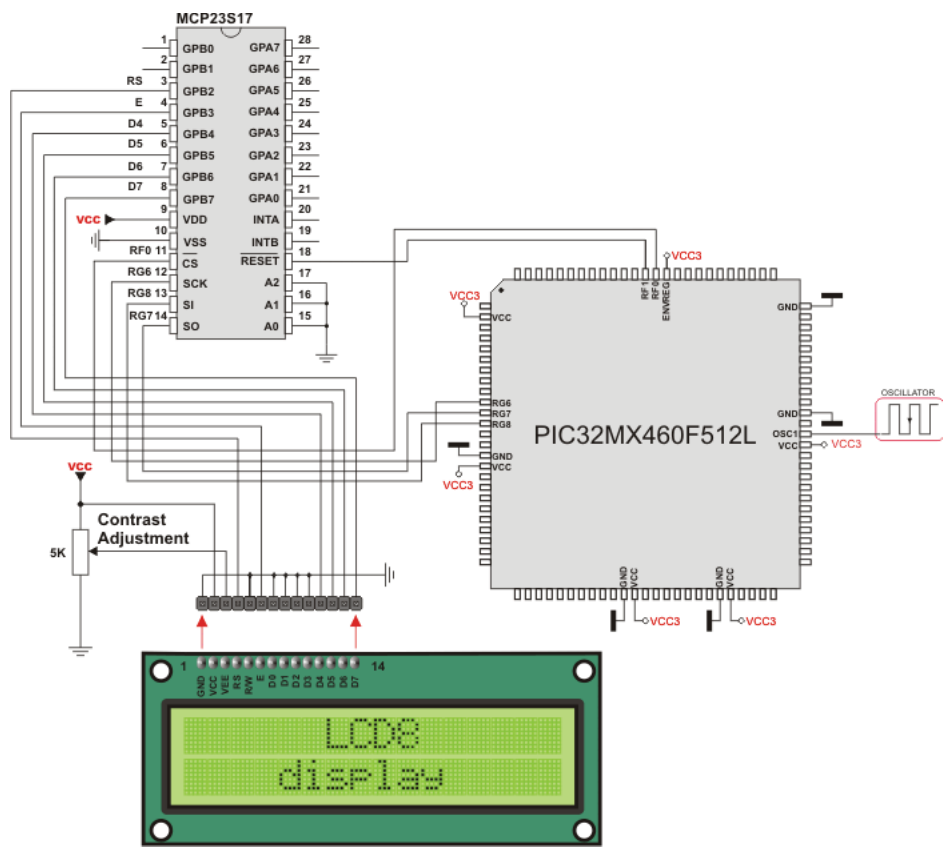
Default Pin Configuration

Use `SPI_Lcd_Init` for default pin settings (see the first figure below).

Copy Code To Clipboard

```
program Spi_Lcd;

var text : array[16] of char;
var counter : byte;

// Port Expander module connections
var SPExpanderRST : sbit at LATD8_bit;
    SPExpanderCS  : sbit at LATD9_bit;
    SPExpanderRST_Direction : sbit at TRISD8_bit;
    SPExpanderCS_Direction  : sbit at TRISD9_bit;
// End Port Expander module connections

procedure Move_Delay();                     // Function used for text moving
  begin
    Delay_ms(500);                          // You can change the moving speed here
  end;

  begin
    text := 'mikroElektronika';
    CHECON := 0x32;
    AD1PCFG := 0xFFFF;                      // Configure AN pins as digital

    // If Port Expander Library uses SPI2 module
    // Initialize SPI module used with PortExpander
      SPI2_Init_Advanced(_SPI_MASTER,_SPI_8_BIT,  4,  _SPI_SS_DISABLE,_SPI_DATA_SAMPLE_
MIDDLE,_SPI_CLK_IDLE_LOW,_SPI_ACTIVE_2_IDLE);

    Spi_Lcd_Config(0);                     // Initialize LCD over SPI interface
    Spi_Lcd_Cmd(_LCD_CLEAR);               // Clear display
    Spi_Lcd_Cmd(_LCD_CURSOR_OFF);          // Turn cursor off
    Spi_Lcd_Out(1,6, 'mikroE');            // Print text to LCD, 1st row, 6th column
    Spi_Lcd_Chr_CP('!');                   // Append '!'
    Spi_Lcd_Out(2,1, text);                // Print text to LCD, 2nd row, 1st column

    //  Spi_Lcd_Out(3,1,'mikroE');         // For LCD with more than two rows
    //  Spi_Lcd_Out(4,15,'mikroE');        // For LCD with more than two rows

    Delay_ms(2000);

    // Moving text
    for counter := 0 to 3 do                // Move text to the right 4 times
      begin
        Spi_Lcd_Cmd(_LCD_SHIFT_RIGHT);
        Move_Delay();
      end;
```

```
while TRUE do                                   // Endless loop
      begin
        for counter := 0 to 6 do               // Move text to the left 7 times
          begin
            Spi_Lcd_Cmd(_LCD_SHIFT_LEFT);
            Move_Delay();
          end;

        for counter := 0 to 6 do               // Move text to the right 7 times
          begin
            Spi_Lcd_Cmd(_LCD_SHIFT_RIGHT);
            Move_Delay();
          end;

      end;
  end.
```



Lcd HW connection by default initialization (using SPI_Lcd_Init)

SPI Lcd8 (8-bit interface) Library

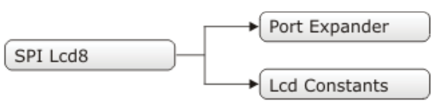
The mikroPascal PRO for PIC32 provides a library for communication with Lcd (with HD44780 compliant controllers) in 8-bit mode via SPI interface.

For creating a custom set of Lcd characters use Lcd Custom Character Tool.

Important:

- When using this library with PIC32 family MCUs be aware of their voltage incompatibility with certain number of Lcd modules.
  So, additional external power supply for these modules may be required.
- Library uses the SPI module for communication. The user must initialize the appropriate SPI module before using the SPI Lcd8 Library.
- For MCUs with multiple SPI modules it is possible to initialize all of them and then switch by using the `SPI_Set_Active()` routine. See the SPI Library functions.
- This Library is designed to work with the mikroElektronika's Serial Lcd/Glcd Adapter Board pinout, see schematic at the bottom of this page for details.

Library Dependency Tree



External dependencies of SPI Lcd Library

The implementation of SPI Lcd Library routines is based on Port Expander Library routines.

External dependencies are the same as Port Expander Library external dependencies.

Library Routines

- SPI_Lcd8_Config
- SPI_Lcd8_Out
- SPI_Lcd8_Out_Cp
- SPI_Lcd8_Chr
- SPI_Lcd8_Chr_Cp
- SPI_Lcd8_Cmd

SPI_Lcd8_Config

Prototype	`procedure SPI_Lcd8_Config(DeviceAddress : byte);`
Description	Initializes the Lcd module via SPI interface.
Parameters	- `DeviceAddress`: SPI expander hardware address, see schematic at the bottom of this page
Returns	Nothing.
Requires	Global variables: - `SPExpanderCS`: Chip Select line - `SPExpanderRST`: Reset line - `SPExpanderCS_Direction`: Direction of the Chip Select pin - `SPExpanderRST_Direction`: Direction of the Reset pin must be defined before using this function. The SPI module needs to be initialized. See SPIx_Init and SPIx_Init_Advanced routines.
Example	see code below
Notes	None.

```
// Port Expander module connections
var SPExpanderRST : sbit at LATF0_bit;
var SPExpanderCS  : sbit at LATF1_bit;
var SPExpanderRST_Direction : sbit at TRISF0_bit;
var SPExpanderCS_Direction  : sbit at TRISF1_bit;
// End Port Expander module connections
...
// If Port Expander Library uses SPI1 module
SPI1_Init();                                 // Initialize SPI module used
with PortExpander
SPI_Lcd8_Config(0);                          // intialize Lcd in 8bit mode
via SPI
```

SPI_Lcd8_Out

Prototype	`procedure SPI_Lcd8_Out(row, column: byte; var text: string);`
Description	Prints text on Lcd starting from specified position. Both string variables and literals can be passed as a text.
Parameters	- `row`: starting position row number - `column`: starting position column number - `text`: text to be written
Returns	Nothing.
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd8_Config routine.
Example	`// Write text "Hello!" on Lcd starting from row 1, column 3:` `SPI_Lcd8_Out(1, 3, 'Hello!');`
Notes	None.

SPI_Lcd8_Out_Cp

Prototype	`procedure SPI_Lcd8_Out_CP(var text: string);`
Description	Prints text on Lcd at current cursor position. Both string variables and literals can be passed as a text.
Parameters	- `text`: text to be written
Returns	Nothing.
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd8_Config routine.
Example	`// Write text "Here!" at current cursor position:` `SPI_Lcd8_Out_Cp('Here!');`
Notes	None.

SPI_Lcd8_Chr

Prototype	`procedure SPI_Lcd8_Chr(row, column, out_char: byte);`
Description	Prints character on Lcd at specified position. Both variables and literals can be passed as character.
Parameters	- `row`: writing position row number - `column`: writing position column number - `out_char`: character to be written
Returns	Nothing.
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd8_Config routine.
Example	`// Write character "i" at row 2, column 3:` `SPI_Lcd8_Chr(2, 3, 'i');`
Notes	None.

SPI_Lcd8_Chr_Cp

Prototype	`procedure SPI_Lcd8_Chr_CP(out_char: byte);`
Description	Prints character on Lcd at current cursor position. Both variables and literals can be passed as character.
Parameters	- `out_char`: character to be written
Returns	Nothing.
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd8_Config routine.

Example	Print “e” at current cursor position:
	
	`// Write character “e” at current cursor position:`
	`SPI_Lcd8_Chr_Cp(‘e’);`

Notes	None.

SPI_Lcd8_Cmd

Prototype	`procedure SPI_Lcd8_Cmd(out_char: byte);`
Description	Sends command to Lcd.
Parameters	- `out_char`: command to be sent
Returns	Nothing.
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd8_Config routine.
Example	`// Clear Lcd display:` `SPI_Lcd8_Cmd(_LCD_CLEAR);`
Notes	Predefined constants can be passed to the routine, see Available SPI Lcd8 Commands.

Available SPI Lcd8 Commands

SPI Lcd8 Command	Purpose
_LCD_FIRST_ROW	Move cursor to the 1st row
_LCD_SECOND_ROW	Move cursor to the 2nd row
_LCD_THIRD_ROW	Move cursor to the 3rd row
_LCD_FOURTH_ROW	Move cursor to the 4th row
_LCD_CLEAR	Clear display
_LCD_RETURN_HOME	Return cursor to home position, returns a shifted display to its original position. Display data RAM is unaffected.
_LCD_CURSOR_OFF	Turn off cursor
_LCD_UNDERLINE_ON	Underline cursor on
_LCD_BLINK_CURSOR_ON	Blink cursor on
_LCD_MOVE_CURSOR_LEFT	Move cursor left without changing display data RAM
_LCD_MOVE_CURSOR_RIGHT	Move cursor right without changing display data RAM
_LCD_TURN_ON	Turn Lcd display on
_LCD_TURN_OFF	Turn Lcd display off
_LCD_SHIFT_LEFT	Shift display left without changing display data RAM
_LCD_SHIFT_RIGHT	Shift display right without changing display data RAM

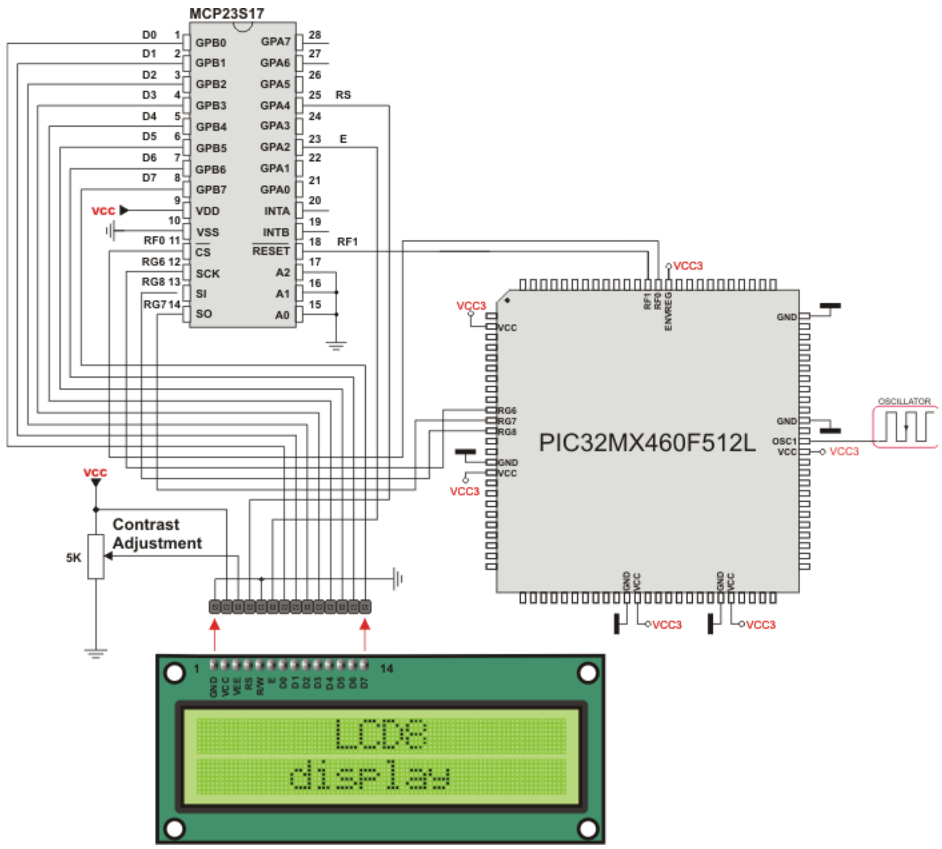
Library Example

This example demonstrates how to communicate Lcd in 8-bit mode via the SPI module, using serial to parallel convertor MCP23S17.

Copy Code To Clipboard

```
program SPI_Lcd8;

var text : array[16] of char;
var counter : byte;

// Port Expander module connections
var SPExpanderRST : sbit at LATD8_bit;
    SPExpanderCS  : sbit at LATD9_bit;
    SPExpanderRST_Direction : sbit at TRISD8_bit;
    SPExpanderCS_Direction  : sbit at TRISD9_bit;
// End Port Expander module connections

procedure Move_Delay();                 // Function used for text moving
  begin
    Delay_ms(500);                      // You can change the moving speed here
  end;

 begin
    CHECON := 0x32;
    AD1PCFG := 0xFFFF;                  // Configure AN pins as digital
    text := 'mikroElektronika';
```

```
    // If Port Expander Library uses SPI2 module
    // Initialize SPI module used with PortExpander
      SPI2_Init_Advanced(_SPI_MASTER,_SPI_8_BIT, 4, _SPI_SS_DISABLE,_SPI_DATA_SAMPLE_
MIDDLE,_SPI_CLK_IDLE_LOW,_SPI_ACTIVE_2_IDLE);

    SPI_Lcd8_Config(0);                       // Initialize Lcd over SPI interface
    SPI_Lcd8_Cmd(_LCD_CLEAR);                 // Clear display
    SPI_Lcd8_Cmd(_LCD_CURSOR_OFF);            // Turn cursor off
    SPI_Lcd8_Out(1,6, 'mikroE');              // Print text to Lcd, 1st row, 6th column
    SPI_Lcd8_Chr_CP('!');                     // Append '!'
    SPI_Lcd8_Out(2,1, text);                  // Print text to Lcd, 2nd row, 1st column

    Delay_ms(2000);

    // Moving text
    for counter := 0 to 3 do                      // Move text to the right 4 times
      begin
        Spi_Lcd8_Cmd(_LCD_SHIFT_RIGHT);
        Move_Delay();
      end;

    while TRUE do                                 // Endless loop
      begin
        for counter := 0 to 6 do                  // Move text to the left 7 times
          begin
            Spi_Lcd8_Cmd(_LCD_SHIFT_LEFT);
            Move_Delay();
          end;

        for counter := 0 to 6 do                  // Move text to the right 7 times
          begin
            Spi_Lcd8_Cmd(_LCD_SHIFT_RIGHT);
            Move_Delay();
          end;

      end;
  end.
```



SPI Lcd8 HW connection

SPI T6963C Graphic Lcd Library

The mikroPascal PRO for PIC32 provides a library for working with Glcds based on TOSHIBA T6963C controller via SPI interface. The Toshiba T6963C is a very popular Lcd controller for the use in small graphics modules. It is capable of controlling displays with a resolution up to 240x128. Because of its low power and small outline it is most suitable for mobile applications such as PDAs, MP3 players or mobile measurement equipment. Although this controller is small, it has a capability of displaying and merging text and graphics and it manages all interfacing signals to the displays Row and Column drivers.

For creating a custom set of Glcd images use Glcd Bitmap Editor Tool.

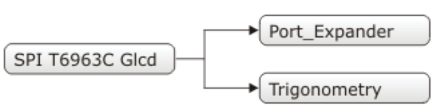
Important:

- When using this library with PIC32 family MCUs be aware of their voltage incompatibility with certain number of T6963C based Glcd modules. So, additional external power supply for these modules may be required.
- Glcd size based initialization routines can be found in setup library files located in the Uses folder.
- The user must make sure that used MCU has appropriate ports and pins. If this is not the case the user should adjust initialization routines.
- The library uses the SPI module for communication. The user must initialize the appropriate SPI module before using the SPI T6963C Glcd Library.
- For MCUs with multiple SPI modules it is possible to initialize both of them and then switch by using the `SPI_Set_Active()` routine. See the SPI Library functions.
- This Library is designed to work with mikroElektronika's Serial Glcd 240x128 and 240x64 Adapter Boards pinout, see schematic at the bottom of this page for details.
- To use constants located in `__Lib_SPIT6963C_Const.mpas` file, user must include it the source file : uses `__Lib_SPIT6963C_Const`;.

Some mikroElektronika's adapter boards have pinout different from T6369C datasheets. Appropriate relations between these labels are given in the table below:

Adapter Board	T6369C datasheet
RS	C/D
R/W	/RD
E	/WR

Library Dependency Tree



External dependencies of SPI T6963C Graphic Lcd Library

The implementation of SPI T6963C Graphic Lcd Library routines is based on Port Expander Library routines.

External dependencies are the same as Port Expander Library external dependencies.

Library Routines

- SPI_T6963C_config
- SPI_T6963C_writeData
- SPI_T6963C_writeCommand
- SPI_T6963C_setPtr
- SPI_T6963C_waitReady
- SPI_T6963C_fill
- SPI_T6963C_dot
- SPI_T6963C_write_char
- SPI_T6963C_Write_Text
- SPI_T6963C_line
- SPI_T6963C_rectangle
- SPI_T6963C_rectangle_round_edges
- SPI_T6963C_rectangle_round_edges_fill
- SPI_T6963C_box
- SPI_T6963C_circle
- SPI_T6963C_circle_fill
- SPI_T6963C_image
- SPI_T6963C_PartialImage
- SPI_T6963C_sprite
- SPI_T6963C_set_cursor
- SPI_T6963C_clearBit
- SPI_T6963C_setBit
- SPI_T6963C_negBit
- SPI_T6963C_displayGrPanel
- SPI_T6963C_displayTxtPanel
- SPI_T6963C_setGrPanel
- SPI_T6963C_setTxtPanel
- SPI_T6963C_panelFill
- SPI_T6963C_grFill
- SPI_T6963C_txtFill
- SPI_T6963C_cursor_height
- SPI_T6963C_graphics
- SPI_T6963C_text
- SPI_T6963C_cursor
- SPI_T6963C_cursor_blink

SPI_T6963C_config

<table>
<tr><td>Prototype</td><td><code>procedure SPI_T6963C_config(width, height, fntW : word; DeviceAddress : byte; wr, rd, cd, rst : byte);</code></td></tr>
<tr><td>Description</td><td>Initializes T6963C Graphic Lcd controller.

Display RAM organization:
The library cuts RAM into panels: a complete panel is one graphics panel followed by a text panel (see schematic below).

<pre>
+---------------------+ /\
+ GRAPHICS PANEL #0   + |
+   + |
+   + |
+   + |
+---------------------+ | PANEL 0
+ TEXT PANEL #0   + |
+   + \/
+---------------------+ /\
+ GRAPHICS PANEL #1   + |
+   + |
+   + |
+   + |
+---------------------+ | PANEL 1
+ TEXT PANEL #1   + |
+   + |
+---------------------+ \/
</pre></td></tr>
<tr><td>Parameters</td><td>- <code>width</code>: width of the Glcd panel
- <code>height</code>: height of the Glcd panel
- <code>fntW</code>: font width
- <code>DeviceAddress</code>: SPI expander hardware address, see schematic at the bottom of this page
- <code>wr</code>: write signal pin on Glcd control port
- <code>rd</code>: read signal pin on Glcd control port
- <code>cd</code>: command/data signal pin on Glcd control port
- <code>rst</code>: reset signal pin on Glcd control port</td></tr>
<tr><td>Returns</td><td>Nothing.</td></tr>
<tr><td>Requires</td><td>Global variables:

- <code>SPExpanderCS</code>: Chip Select line
- <code>SPExpanderRST</code>: Reset line
- <code>SPExpanderCS_Direction</code>: Direction of the Chip Select pin
- <code>SPExpanderRST_Direction</code>: Direction of the Reset pin

must be defined before using this function.

The SPI module needs to be initialized. See the SPIx_Init and SPIx_Init_Advanced routines.</td></tr>
</table>

Example	`// Port Expander module connections`
	`   SPExpanderRST : sbit at LATF0_bit;`
	`   SPExpanderCS  : sbit at LATF1_bit;`
	`   SPExpanderRST_Direction : sbit at TRISF0_bit;`
	`   SPExpanderCS_Direction  : sbit at TRISF1_bit;`
	`// End Port Expander module connections`
	`...`
	`// Initialize SPI module`
	`SPI1_Init();`
	`SPI_T6963C_config(240, 64, 8, 0, 0, 1, 3, 4);`

Notes	None.

SPI_T6963C_writeData

Prototype	`procedure SPI_T6963C_writeData(data_ : byte);`
Description	Writes data to T6963C controller via SPI interface.
Parameters	- `data_`: data to be written
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_writeData(data_);`
Notes	None.

SPI_T6963C_writeCommand

Prototype	`procedure SPI_T6963C_writeCommand(data_ : byte);`
Description	Writes command to T6963C controller via SPI interface.
Parameters	- `data_`: command to be written
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_writeCommand(SPI_T6963C_CURSOR_POINTER_SET);`
Notes	None.

SPI_T6963C_setPtr

Prototype	`procedure SPI_T6963C_setPtr(p : word; c : byte);`
Description	Sets the memory pointer `p` for command `p`.
Parameters	- `p`: address where command should be written - `c`: command to be written
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_setPtr(SPI_T6963C_grHomeAddr + start, SPI_T6963C_ADDRESS_POINTER_SET);`
Notes	None.

SPI_T6963C_waitReady

Prototype	`procedure SPI_T6963C_waitReady();`
Description	Pools the status byte, and loops until Toshiba Glcd module is ready.
Parameters	None.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_waitReady();`
Notes	None.

SPI_T6963C_fill

Prototype	`procedure SPI_T6963C_fill(v : byte; start, len : word);`
Description	Fills controller memory block with given byte.
Parameters	- `v`: byte to be written - `start`: starting address of the memory block - `len`: length of the memory block in bytes
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_fill(0x33,0x00FF,0x000F);`
Notes	None.

SPI_T6963C_dot

Prototype	`procedure SPI_T6963C_dot(x, y : integer; color : byte);`
Description	Writes a char in the current text panel of Glcd at coordinates (`x`, `y`).
Returns	- `x`: dot position on x-axis - `y`: dot position on y-axis - `color`: color parameter. Valid values: SPI_T6963C_BLACK and SPI_T6963C_WHITE
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_dot(x0, y0, SPI_T6963C_BLACK);`
Notes	None.

SPI_T6963C_write_char

Prototype	`procedure SPI_T6963C_write_char(c, x, y, mode : byte);`
Description	Writes a char in the current text panel of Glcd at coordinates (x, y).
Parameters	- `c`: char to be written - `x`: char position on x-axis - `y`: char position on y-axis - `mode`: mode parameter. Valid values: SPI_T6963C_ROM_MODE_OR, SPI_T6963C_ROM_MODE_XOR, SPI_T6963C_ROM_MODE_AND and SPI_T6963C_ROM_MODE_TEXT Mode parameter explanation: - OR Mode: In the OR-Mode, text and graphics can be displayed and the data is logically “OR-ed”. This is the most common way of combining text and graphics for example labels on buttons. - XOR-Mode: In this mode, the text and graphics data are combined via the logical “exclusive OR”. This can be useful to display text in negative mode, i.e. white text on black background. - AND-Mode: The text and graphic data shown on display are combined via the logical “AND function”. - TEXT-Mode: This option is only available when displaying just a text. The Text Attribute values are stored in the graphic area of display memory. For more details see the T6963C datasheet.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_write_char('A',22,23,SPI_T6963C_ROM_MODE_AND);`
Notes	None.

SPI_T6963C_write_text

Prototype	`procedure SPI_T6963C_write_text(var str : array[10] of byte; x, y, mode : byte);`
Description	Writes text in the current text panel of Glcd at coordinates (x, y).
Parameters	- `str`: text to be written - `x`: text position on x-axis - `y`: text position on y-axis - `mode`: mode parameter. Valid values: SPI_T6963C_ROM_MODE_OR, SPI_T6963C_ROM_MODE_XOR, SPI_T6963C_ROM_MODE_AND and SPI_T6963C_ROM_MODE_TEXT Mode parameter explanation: - OR Mode: In the OR-Mode, text and graphics can be displayed and the data is logically “OR-ed”. This is the most common way of combining text and graphics for example labels on buttons. - XOR-Mode: In this mode, the text and graphics data are combined via the logical “exclusive OR”. This can be useful to display text in negative mode, i.e. white text on black background. -AND-Mode: The text and graphic data shown on the display are combined via the logical “AND function”. - TEXT-Mode: This option is only available when displaying just a text. The Text Attribute values are stored in the graphic area of display memory. For more details see the T6963C datasheet.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_write_text('GLCD LIBRARY DEMO, WELCOME !', 0, 0, SPI_T6963C_ROM_MODE_XOR);`
Notes	None.

SPI_T6963C_line

Prototype	`procedure SPI_T6963C_line(x0, y0, x1, y1 : integer; pcolor : byte);`
Description	Draws a line from (x0, y0) to (x1, y1).
Parameters	- `x0`: x coordinate of the line start - `y0`: y coordinate of the line end - `x1`: x coordinate of the line start - `y1`: y coordinate of the line end - `pcolor`: color parameter. Valid values: SPI_T6963C_BLACK and SPI_T6963C_WHITE
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_line(0, 0, 239, 127, SPI_T6963C_WHITE);`
Notes	None.

SPI_T6963C_rectangle

Prototype	`procedure SPI_T6963C_rectangle(x0, y0, x1, y1 : integer; pcolor : byte);`
Description	Draws a rectangle on Glcd.
Parameters	- `x0`: x coordinate of the upper left rectangle corner - `y0`: y coordinate of the upper left rectangle corner - `x1`: x coordinate of the lower right rectangle corner - `y1`: y coordinate of the lower right rectangle corner - `pcolor`: color parameter. Valid values: SPI_T6963C_BLACK and SPI_T6963C_WHITE
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_rectangle(20, 20, 219, 107, SPI_T6963C_WHITE);`
Notes	None.

SPI_T6963C_rectangle_round_edges

Prototype	`procedure SPI_T6963C_rectangle_round_edges(x0 : integer; y0 : integer; x1 : integer; y1 : integer; radius : integer; pcolor : byte);`
Description	Draws a rounded edge rectangle on Glcd.
Parameters	- `x0`: x coordinate of the upper left rectangle corner - `y0`: y coordinate of the upper left rectangle corner - `x1`: x coordinate of the lower right rectangle corner - `y1`: y coordinate of the lower right rectangle corner - `round_radius`: radius of the rounded edge. - `pcolor`: color parameter. Valid values: SPI_T6963C_BLACK and SPI_T6963C_WHITE
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_rectangle_round_edges(20, 20, 219, 107, 12, SPI_T6963C_WHITE);`
Notes	None.

SPI_T6963C_rectangle_round_edges_fill

Prototype	`procedure SPI_T6963C_rectangle_round_edges_fill(x0 : integer; y0 : integer; x1 : integer; y1 : integer; radius : integer; pcolor : byte);`
Description	Draws a filled rounded edge rectangle on Glcd.
Parameters	- `x0`: x coordinate of the upper left rectangle corner - `y0`: y coordinate of the upper left rectangle corner - `x1`: x coordinate of the lower right rectangle corner - `y1`: y coordinate of the lower right rectangle corner - `round_radius`: radius of the rounded edge - `pcolor`: color parameter. Valid values: `SPI_T6963C_BLACK` and `SPI_T6963C_WHITE`
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_rectangle_round_edges_fill(20, 20, 219, 107, 12, SPI_T6963C_WHITE);`
Notes	None.

SPI_T6963C_box

Prototype	`procedure SPI_T6963C_box(x0, y0, x1, y1 : integer; pcolor : byte);`
Description	Draws a box on the Glcd
Parameters	- `x0`: x coordinate of the upper left box corner - `y0`: y coordinate of the upper left box corner - `x1`: x coordinate of the lower right box corner - `y1`: y coordinate of the lower right box corner - `pcolor`: color parameter. Valid values: SPI_T6963C_BLACK and SPI_T6963C_WHITE
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_box(0, 119, 239, 127, SPI_T6963C_WHITE);`
Notes	None.

SPI_T6963C_circle

Prototype	`procedure SPI_T6963C_circle(x, y : integer; r : longint; pcolor : word);`
Description	Draws a circle on the Glcd.
Parameters	- `x`: x coordinate of the circle center - `y`: y coordinate of the circle center - `r`: radius size - `pcolor`: color parameter. Valid values: SPI_T6963C_BLACK and SPI_T6963C_WHITE
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_circle(120, 64, 110, SPI_T6963C_WHITE);`
Notes	None.

SPI_T6963C_circle_fill

Prototype	`procedure SPI_T6963C_circle_fill(x : integer; y : integer; r : longint; pcolor : word);`
Description	Draws a filled circle on the Glcd.
Parameters	- `x`: x coordinate of the circle center - `y`: y coordinate of the circle center - `r`: radius size - `pcolor`: color parameter. Valid values: `SPI_T6963C_BLACK` and `SPI_T6963C_WHITE`
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_circle_fill(120, 64, 110, SPI_T6963C_WHITE);`
Notes	None.

SPI_T6963C_image

Prototype	`procedure SPI_T6963C_image(pic : ^ const byte);`
Description	Displays bitmap on Glcd.
Parameters	- `pic`: image to be displayed. Bitmap array can be located in both code and RAM memory (due to the mikroPascal PRO for PIC32 pointer to const and pointer to RAM equivalency).
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_image(my_image);`
Notes	Image dimension must match the display dimension. Use the integrated Glcd Bitmap Editor (menu option **Tools › Glcd Bitmap Editor**) to convert image to a constant array suitable for displaying on Glcd.

SPI_T6963C_PartialImage

Prototype	`procedure SPI_T6963C_PartialImage(x_left, y_top, width, height, picture_width, picture_height : word; const image : ^byte);`
Description	Displays a partial area of the image on a desired location.
Parameters	- `x_left`: x coordinate of the desired location (upper left coordinate). - `y_top`: y coordinate of the desired location (upper left coordinate). - `width`: desired image width. - `height`: desired image height. - `picture_width`: width of the original image. - `picture_height`: height of the original image. - `image`: image to be displayed. Bitmap array can be located in both code and RAM memory (due to the mikroPascal PRO for PIC pointer to const and pointer to RAM equivalency).
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`// Draws a 10x15 part of the image starting from the upper left corner on the coordinate (10,12). Original image size is 16x32.` `SPI_T6963C_PartialImage(10, 12, 10, 15, 16, 32, @image);`
Notes	Use the integrated Glcd Bitmap Editor (menu option **Tools › Glcd Bitmap Editor**) to convert image to a constant array suitable for displaying on Glcd.

SPI_T6963C_sprite

Prototype	`procedure SPI_T6963C_sprite(px, py : byte; const pic : ^byte; sx, sy : byte);`
Description	Fills graphic rectangle area (px, py) to (px+sx, py+sy) with custom size picture.
Parameters	- `px`: x coordinate of the upper left picture corner. Valid values: multiples of the font width - `py`: y coordinate of the upper left picture corner - `pic`: picture to be displayed - `sx`: picture width. Valid values: multiples of the font width - `sy`: picture height
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_sprite(76, 4, einstein, 88, 119); // draw a sprite`
Notes	If `px` and `sx` parameters are not multiples of the font width they will be scaled to the nearest lower number that is a multiple of the font width.

SPI_T6963C_set_cursor

Prototype	`procedure SPI_T6963C_set_cursor(x, y : byte);`
Description	Sets cursor to row x and column y.
Parameters	- `x`: cursor position row number - `y`: cursor position column number
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_set_cursor(cposx, cposy);`
Notes	None.

SPI_T6963C_clearBit

Prototype	`procedure SPI_T6963C_clearBit(b : byte);`
Description	Clears control port bit(s).
Parameters	- `b`: bit mask. The function will clear bit `x` on control port if bit `x` in bit mask is set to `1`.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`// clear bits 0 and 1 on control port` `SPI_T6963C_clearBit(0x03);`
Notes	None.

SPI_T6963C_setBit

Prototype	`procedure SPI_T6963C_setBit(b : byte);`
Description	Sets control port bit(s).
Parameters	- `b`: bit mask. The function will set bit `x` on control port if bit `x` in bit mask is set to `1`.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`// set bits 0 and 1 on control port` `SPI_T6963C_setBit(0x03);`
Notes	None.

SPI_T6963C_negBit

Prototype	`procedure SPI_T6963C_negBit(b : byte);`
Description	Negates control port bit(s).
Parameters	- `b`: bit mask. The function will negate bit `x` on control port if bit `x` in bit mask is set to `1`.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`// negate bits 0 and 1 on control port` `SPI_T6963C_negBit(0x03);`
Notes	None.

SPI_T6963C_displayGrPanel

Prototype	`procedure SPI_T6963C_displayGrPanel(n : word);`
Description	Display selected graphic panel.
Parameters	- `n`: graphic panel number. Valid values: `0` and `1`.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`// display graphic panel 1` `SPI_T6963C_displayGrPanel(1);`
Notes	None.

SPI_T6963C_displayTxtPanel

Prototype	`procedure SPI_T6963C_displayTxtPanel(n : word);`
Description	Display selected text panel.
Parameters	- `n`: text panel number. Valid values: `0` and `1`.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`// display text panel 1` `SPI_T6963C_displayTxtPanel(1);`
Notes	None.

SPI_T6963C_setGrPanel

Prototype	`procedure SPI_T6963C_setGrPanel(n : word);`
Description	Compute start address for selected graphic panel and set appropriate internal pointers. All subsequent graphic operations will be preformed at this graphic panel.
Parameters	- `n`: graphic panel number. Valid values: `0` and `1`.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`// set graphic panel 1 as current graphic panel.` `SPI_T6963C_setGrPanel(1);`
Notes	None.

SPI_T6963C_setTxtPanel

Prototype	`procedure SPI_T6963C_setTxtPanel(n : word);`
Description	Compute start address for selected text panel and set appropriate internal pointers. All subsequent text operations will be preformed at this text panel.
Parameters	- `n`: text panel number. Valid values: `0` and `1`.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`// set text panel 1 as current text panel.` `SPI_T6963C_setTxtPanel(1);`
Notes	None.

SPI_T6963C_panelFill

Prototype	`procedure SPI_T6963C_panelFill(v : word);`
Description	Fill current panel in full (graphic+text) with appropriate value (0 to clear).
Parameters	- `v`: value to fill panel with.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`clear current panel` `SPI_T6963C_panelFill(0);`
Notes	None.

SPI_T6963C_grFill

Prototype	`procedure SPI_T6963C_grFill(v: word);`
Description	Fill current graphic panel with appropriate value (0 to clear).
Parameters	- `v`: value to fill graphic panel with.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`// clear current graphic panel` `SPI_T6963C_grFill(0);`
Notes	None.

SPI_T6963C_txtFill

Prototype	`procedure SPI_T6963C_txtFill(v : word);`
Description	Fill current text panel with appropriate value (0 to clear).
Parameters	- `v`: this value increased by 32 will be used to fill text panel.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`// clear current text panel` `SPI_T6963C_txtFill(0);`
Notes	None.

SPI_T6963C_cursor_height

Prototype	`procedure SPI_T6963C_cursor_height(n: byte);`
Description	Set cursor size.
Parameters	- `n`: cursor height. Valid values: `0..7`.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_cursor_height(7);`
Notes	None.

SPI_T6963C_graphics

Prototype	`procedure SPI_T6963C_graphics(n : word);`
Description	Enable/disable graphic displaying.
Parameters	- `n`: graphic enable/disable parameter. Valid values: `0` (disable graphic dispaying) and `1` (enable graphic displaying).
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`// enable graphic displaying` `SPI_T6963C_graphics(1);`
Notes	None.

SPI_T6963C_text

Prototype	`procedure SPI_T6963C_text(n : word);`
Description	Enable/disable text displaying.
Parameters	- `n`: text enable/disable parameter. Valid values: `0` (disable text dispaying) and `1` (enable text displaying).
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`// enable text displaying` `SPI_T6963C_text(1);`
Notes	None.

SPI_T6963C_cursor

Prototype	`procedure SPI_T6963C_cursor(n : word);`
Description	Set cursor on/off.
Parameters	- `n`: on/off parameter. Valid values: `0` (set cursor off) and `1` (set cursor on).
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`// set cursor on` `SPI_T6963C_cursor(1);`
Notes	None.

SPI_T6963C_cursor_blink

Prototype	`procedure SPI_T6963C_cursor_blink(n : word);`
Description	Enable/disable cursor blinking.
Parameters	- `n`: cursor blinking enable/disable parameter. Valid values: `0` (disable cursor blinking) and `1` (enable cursor blinking).
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`// enable cursor blinking` `SPI_T6963C_cursor_blink(1);`
Notes	None.

Library Example

The following drawing demo tests advanced routines of the SPI T6963C Glcd library. Hardware configurations in this example are made for the LV-32MX v6 board and PIC32MX460F512L.

Copy Code To Clipboard

```
program SPI_T6963C_240x128;

uses __Lib_SPIT6963C_Const;

var
// Port Expander module connections
   SPExpanderRST : sbit at LATD8_bit;  // for writing to output pin always use latch
   SPExpanderCS  : sbit at LATD9_bit;  // for writing to output pin always use latch
   SPExpanderRST_Direction : sbit at TRISD8_bit;
   SPExpanderCS_Direction  : sbit at TRISD9_bit;
// End Port Expander module connections

var   panel : byte;         // current panel
          i : word;         // general purpose register
      curs  : byte;         // cursor visibility
      cposx,
      cposy : word;         // cursor x-y position
      txt, txt1 : string[29];
      txt2 : string[21];
```

```
begin
  {$DEFINE COMPLETE_EXAMPLE}
  {$DEFINE LINE_DEMO}           // Uncomment to demonstrate line drawing routines
  {$DEFINE FILL_DEMO}           // Uncomment to demonstrate fill routines
  {$DEFINE PARTIAL_IMAGE_DEMO}  // Uncomment to demonstrate partial image routine

  txt1 := ' EINSTEIN WOULD HAVE LIKED mE';
  txt  := ' GLCD LIBRARY DEMO, WELCOME !';
  txt2 := ' Partial image demo! ';

  CHECON := 0x32;
  AD1PCFG := 0xFFFF;            // initialize AN pins as digital

  TRISB8_bit  := 1;             // Set RB8 as input
  TRISB9_bit  := 1;             // Set RB9 as input
  TRISB10_bit := 1;             // Set RB10 as input
  TRISB11_bit := 1;             // Set RB11 as input
  TRISB12_bit := 1;             // Set RB12 as input
  TRISB13_bit := 1;             // Set RB13 as input

  {*
  * init display for 240 pixel width and 128 pixel height
  * 8 bits character width
  * data bus on MCP23S17 portB
  * control bus on MCP23S17 portA
  * bit 2 is !WR
  * bit 1 is !RD
  * bit 0 is !CD
  * bit 4 is RST
  * chip enable, reverse on, 8x8 font internaly set in library
  *}

  // If Port Expander Library uses SPI2 module
  // Initialize SPI module used with PortExpander
     SPI2_Init_Advanced(_SPI_MASTER,_SPI_8_BIT,  4,  _SPI_SS_DISABLE,_SPI_DATA_SAMPLE_
MIDDLE,_SPI_CLK_IDLE_LOW,_SPI_ACTIVE_2_IDLE);

  {*
   * init display for 240 pixel width and 128 pixel height
   * 8 bits character width
   * data bus on MCP23S17 portB
   * control bus on MCP23S17 portA
   * bit 2 is !WR
   * bit 1 is !RD
   * bit 0 is !CD
   * bit 4 is RST
   * chip enable, reverse on, 8x8 font internaly set in library
   *}

  // Initialize SPI Toshiba 240x128
  SPI_T6963C_Config(240, 128, 8, 0, 2, 1, 0, 4);
  //Delay_ms(1000);
```

```
{*
 * Enable both graphics and text display at the same time
 *}

SPI_T6963C_graphics(1);

SPI_T6963C_text(1);

panel := 0;
i := 0;
curs := 0;
cposx := 0;
cposy := 0;

{*
 * Text messages
 *}
SPI_T6963C_write_text(txt, 0, 0, SPI_T6963C_ROM_MODE_XOR);
SPI_T6963C_write_text(txt1, 0, 15, SPI_T6963C_ROM_MODE_XOR);

{*
 * Cursor
 *}
SPI_T6963C_cursor_height(8);        // 8 pixel height
SPI_T6963C_set_cursor(0, 0) ;       // move cursor to top left
SPI_T6963C_cursor(0) ;              // cursor off

{*
 * Draw solid boxes
 *}
SPI_T6963C_box(0, 0, 239, 8, SPI_T6963C_WHITE) ;
SPI_T6963C_box(0, 119, 239, 127, SPI_T6963C_WHITE) ;

{*
 * Draw rectangles
 *}
{$IFDEF LINE_DEMO}
SPI_T6963C_rectangle(0, 0, 239, 127, SPI_T6963C_WHITE);
SPI_T6963C_rectangle(20, 20, 219, 107, SPI_T6963C_WHITE);
SPI_T6963C_rectangle(40, 40, 199, 87, SPI_T6963C_WHITE);
SPI_T6963C_rectangle(60, 60, 179, 67, SPI_T6963C_WHITE);
{$ENDIF}

{*
 * Draw rounded edge rectangle
 *}
{$IFDEF LINE_DEMO}
  SPI_T6963C_Rectangle_Round_Edges(10, 10, 229, 117, 12, SPI_T6963C_WHITE);
  SPI_T6963C_Rectangle_Round_Edges(30, 30, 209, 97, 12, SPI_T6963C_WHITE);
  SPI_T6963C_Rectangle_Round_Edges(50, 50, 189, 77, 12, SPI_T6963C_WHITE);
{$ENDIF}
```

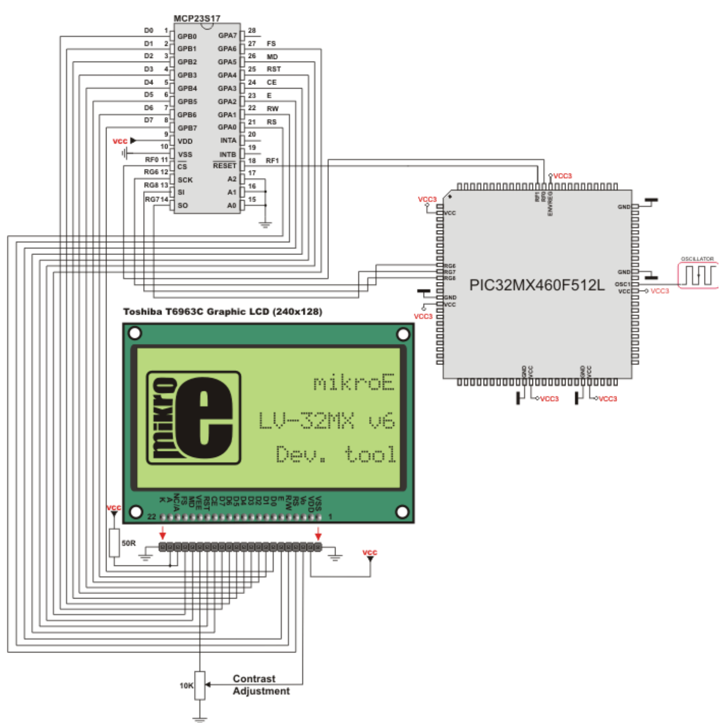
```
{*
 * Draw filled rounded edge rectangle
 *}
{$IFDEF FILL_DEMO}
  SPI_T6963C_Rectangle_Round_Edges_Fill(10, 10, 229, 117, 12, SPI_T6963C_WHITE);
  SPI_T6963C_Rectangle_Round_Edges_Fill(20, 20, 219, 107, 12, SPI_T6963C_BLACK);
  SPI_T6963C_Rectangle_Round_Edges_Fill(30, 30, 209, 97, 12, SPI_T6963C_WHITE);
  SPI_T6963C_Rectangle_Round_Edges_Fill(40, 40, 199, 87, 12, SPI_T6963C_BLACK);
  SPI_T6963C_Rectangle_Round_Edges_Fill(50, 50, 189, 77, 12, SPI_T6963C_WHITE);
{$ENDIF}

{*
 * Draw a cross
 *}
{$IFDEF LINE_DEMO}
  SPI_T6963C_line(0, 0, 239, 127, SPI_T6963C_WHITE);
  SPI_T6963C_line(0, 127, 239, 0, SPI_T6963C_WHITE);
{$ENDIF}

{*
 * Draw circles
 *}
{$IFDEF LINE_DEMO}
SPI_T6963C_circle(120, 64, 10, SPI_T6963C_WHITE);
SPI_T6963C_circle(120, 64, 30, SPI_T6963C_WHITE);
SPI_T6963C_circle(120, 64, 50, SPI_T6963C_WHITE);
SPI_T6963C_circle(120, 64, 70, SPI_T6963C_WHITE);
SPI_T6963C_circle(120, 64, 90, SPI_T6963C_WHITE);
SPI_T6963C_circle(120, 64, 110, SPI_T6963C_WHITE);
SPI_T6963C_circle(120, 64, 130, SPI_T6963C_WHITE);
{$ENDIF}

{*
 * Draw filled circles
 *}
{$IFDEF FILL_DEMO}
  SPI_T6963C_circle_fill(120, 64, 60, SPI_T6963C_WHITE);
  SPI_T6963C_circle_fill(120, 64, 55, SPI_T6963C_BLACK);
  SPI_T6963C_circle_fill(120, 64, 50, SPI_T6963C_WHITE);
  SPI_T6963C_circle_fill(120, 64, 45, SPI_T6963C_BLACK);
  SPI_T6963C_circle_fill(120, 64, 40, SPI_T6963C_WHITE);
  SPI_T6963C_circle_fill(120, 64, 35, SPI_T6963C_BLACK);
  SPI_T6963C_circle_fill(120, 64, 30, SPI_T6963C_WHITE);
  SPI_T6963C_circle_fill(120, 64, 25, SPI_T6963C_BLACK);
  SPI_T6963C_circle_fill(120, 64, 20, SPI_T6963C_WHITE);
  SPI_T6963C_circle_fill(120, 64, 15, SPI_T6963C_BLACK);
  SPI_T6963C_circle_fill(120, 64, 10, SPI_T6963C_WHITE);
  SPI_T6963C_circle_fill(120, 64, 5, SPI_T6963C_BLACK);
{$ENDIF}

Delay_ms(1000);
SPI_T6963C_sprite(76, 4, @einstein_bmp, 88, 119);   // Draw a sprite
Delay_ms(1000);

SPI_T6963C_setGrPanel(1);                            // Select other graphic panel
```

```
  SPI_T6963C_image(@mikroE_240x128_bmp);
  SPI_T6963C_displayGrPanel(1);
  Delay_ms(1000);
  {$IFDEF PARTIAL_IMAGE_DEMO}
    SPI_T6963C_grFill(0);
     SPI_T6963C_PartialImage(0, 0, 64, 64, 240, 128, @mikroE_240x128_bmp);  // Display
partial image
    Delay_ms(1000);
    SPI_T6963C_graphics(0);
  {$ENDIF}
  SPI_T6963C_image(@mikroE_240x128_bmp);
  SPI_T6963C_graphics(1);
  SPI_T6963C_displayGrPanel(0);

  while (TRUE) do                                       // Endless loop
    begin

      {*
       * If RB8 is pressed, toggle the display between graphic panel 0 and graphic 1
       *}
      if(RB8_bit <> 0) then
        begin
          Inc(panel) ;
          panel := panel and 1;
             SPI_T6963C_setPtr((SPI_T6963C_grMemSize + SPI_T6963C_txtMemSize) * panel,
SPI_T6963C_GRAPHIC_HOME_ADDRESS_SET) ;
          Delay_ms(300);
        end

      {*
       * If RB9 is pressed, display only graphic panel
       *}
      else
        if (RB9_bit <> 0) then
          begin
            SPI_T6963C_graphics(1) ;
            SPI_T6963C_text(0) ;
            Delay_ms(300);
          end

      {*
       * If RB10 is pressed, display only text panel
       *}
        else
          if (RB10_bit <> 0) then
            begin
              SPI_T6963C_graphics(0);
              SPI_T6963C_text(1);
              Delay_ms(300);
            end


      {*
       * If RB11 is pressed, display text and graphic panels
       *}
```

```
        else
         if (RB11_bit <> 0) then
           begin
             SPI_T6963C_graphics(1);
             SPI_T6963C_text(1);
             Delay_ms(300);
           end

    {*
     * If RB12 is pressed, change cursor
     *}
         else
           if(RB12_bit <> 0) then
             begin
               Inc(curs);
               if (curs = 3) then
                 curs := 0;
               case curs of
                 0:
                    // no cursor
                    SPI_T6963C_cursor(0);

                 1: begin
                      // blinking cursor
                      SPI_T6963C_cursor(1);
                      SPI_T6963C_cursor_blink(1);
                    end;
                 2: begin
                      // non blinking cursor
                      SPI_T6963C_cursor(1);
                      SPI_T6963C_cursor_blink(0);
                    end;
               end;
               Delay_ms(300);
             end
    {*
     * If RB13 is pressed, perform the "Partial image" demonstration
     *}
    else
      {$IFDEF PARTIAL_IMAGE_DEMO}
        if( RB13_bit = 1) then
          begin
            SPI_T6963C_setGrPanel(0);
            SPI_T6963C_setTxtPanel(0);
            SPI_T6963C_txtFill(0);
            SPI_T6963C_setGrPanel(1);
            SPI_T6963C_setTxtPanel(0);
            SPI_T6963C_graphics(1);
            SPI_T6963C_text(1);
            SPI_T6963C_displayGrPanel(1);
            SPI_T6963C_write_text(txt2, 5, 15, SPI_T6963C_ROM_MODE_XOR);
            Delay_1sec();

            SPI_T6963C_grFill(0);
            SPI_T6963C_PartialImage(0, 0, 64, 64, 240, 128, @mikroE_240x128_bmp);
            Delay_ms(1500);
```

```
              SPI_T6963C_PartialImage(0, 0, 128, 128, 240, 128, @mikroE_240x128_bmp);
              Delay_ms(1500);

              SPI_T6963C_PartialImage(0, 0, 240, 128, 240, 128, @mikroE_240x128_bmp);
              Delay_ms(1500);

              SPI_T6963C_txtFill(0);
              SPI_T6963C_write_text(txt, 0, 0, @mikroE_240x128_bmp);
              SPI_T6963C_write_text(txt1, 0, 15, @mikroE_240x128_bmp);
            end;
        {$ENDIF}
      {*
       * Move cursor, even if not visible
       *}
      Inc(cposx);
      if (cposx = SPI_T6963C_txtCols) then
        begin
          cposx := 0;
          Inc(cposy);
          if (cposy = SPI_T6963C_grHeight / SPI_T6963C_CHARACTER_HEIGHT) then
            cposy := 0;
        end;
      SPI_T6963C_set_cursor(cposx, cposy);

      Delay_ms(100);
    end;
end.
```

HW Connection



SPI T6963C Glcd HW connection

T6963C Graphic Lcd Library

The mikroPascal PRO for PIC32 provides a library for working with Glcds based on TOSHIBA T6963C controller. The Toshiba T6963C is a very popular Lcd controller for the use in small graphics modules. It is capable of controlling displays with a resolution up to 240x128. Because of its low power and small outline it is most suitable for mobile applications such as PDAs, MP3 players or mobile measurement equipment. Although small, this contoller has a capability of displaying and merging text and graphics and it manages all the interfacing signals to the displays Row and Column drivers.

For creating a custom set of Glcd images use Glcd Bitmap Editor Tool.

Important:

- When using this library with PIC32 family of MCUs be aware of their voltage incompatibility with certain number of T6963C based Glcd modules. So, additional external power supply for these modules may be required.
- ChipEnable(CE), FontSelect(FS) and Reverse(MD) have to be set to appropriate levels by the user outside of the T6963C_Init() function. See the Library Example code at the bottom of this page.
- Glcd size based initialization routines can be found in setup library files located in the Uses folder.
- The user must make sure that used MCU has appropriate ports and pins. If this is not the case the user should adjust initialization routines.

Some mikroElektronika's adapter boards have pinout different from T6369C datasheets. Appropriate relations between these labels are given in the table below:

Adapter Board	T6369C datasheet
RS	C/D
R/W	/RD
E	/WR

Library Dependency Tree

External dependencies of T6963C Graphic Lcd Library

The following variables must be defined in all projects using T6963C Graphic Lcd library:	Description:	Example:
`var T6963C_dataPort : word; sfr; external;`	T6963C Data Port.	`var T6963C_dataPort : word at PORTB;`
`var T6963C_ctrlwr : sbit; sfr; external;`	Write signal.	`var T6963C_ctrlwr : sbit at LATF2_bit;`
`var T6963C_ctrlrd : sbit; sfr; external;`	Read signal.	`var T6963C_ctrlrd : sbit at LATF1_bit;`
`var T6963C_ctrlcd : sbit; sfr; external;`	Command/Data signal.	`var T6963C_ctrlcd : sbit at LATF0_bit;`
`var T6963C_ctrlrst : sbit; sfr; external;`	Reset signal.	`var T6963C_ctrlrst : sbit at LATF4_bit;`
`var T6963C_ctrlwr_Direction : sbit; sfr; external;`	Direction of the Write pin.	`var T6963C_ctrlwr_Direction : sbit at TRISF2_bit;`
`var T6963C_ctrlrd_Direction : sbit; sfr; external;`	Direction of the Read pin.	`var T6963C_ctrlrd_Direction : sbit at TRISF1_bit;`
`var T6963C_ctrlcd_Direction : sbit; sfr; external;`	Direction of the Command/ Data pin.	`var T6963C_ctrlcd_Direction : sbit at TRISF0_bit;`
`var T6963C_ctrlrst_Direction : sbit; sfr; external;`	Direction of the Reset pin.	`var T6963C_ctrlrst_Direction : sbit at TRISF4_bit;`

Library Routines

- T6963C_init
- T6963C_writeData
- T6963C_writeCommand
- T6963C_setPtr
- T6963C_waitReady
- T6963C_fill
- T6963C_dot
- T6963C_write_char
- T6963C_write_text
- T6963C_line
- T6963C_rectangle
- T6963C_Rectangle_round_edges
- T6963C_Rectangle_round_edges_fill
- T6963C_box
- T6963C_circle
- T6963C_Circle_fill
- T6963C_image
- T6963C_PartialImage
- T6963C_sprite
- T6963C_set_cursor
- T6963C_displayGrPanel
- T6963C_displayTxtPanel
- T6963C_setGrPanel
- T6963C_setTxtPanel
- T6963C_panelFill
- T6963C_grFill
- T6963C_txtFill
- T6963C_cursor_height
- T6963C_graphics
- T6963C_text
- T6963C_cursor
- T6963C_cursor_blink

T6963C_init

<table>
<tr><td>Prototype</td><td><code>procedure T6963C_init(width, height, fntW : word);</code></td></tr>
<tr><td>Description</td><td>Initalizes the Graphic Lcd controller.

Display RAM organization:
The library cuts the RAM into panels: a complete panel is one graphics panel followed by a text panel (see schematic below).

<pre>
+---------------------+ /\
+ GRAPHICS PANEL #0   + |
+   + |
+   + |
+   + |
+---------------------+ | PANEL 0
+ TEXT PANEL #0   + |
+   + \/
+---------------------+ /\
+ GRAPHICS PANEL #1   + |
+   + |
+   + |
+   + |
+---------------------+ | PANEL 1
+ TEXT PANEL #1   + |
+   + |
+---------------------+ \/
</pre></td></tr>
<tr><td>Parameters</td><td>- <code>width</code>: width of the Glcd panel
- <code>height</code>: height of the Glcd panel
- <code>fntW</code>: font width</td></tr>
<tr><td>Returns</td><td>Nothing.</td></tr>
<tr><td>Requires</td><td>Global variables:

- <code>T6963C_dataPort</code>: Data Port
- <code>T6963C_ctrlwr</code>: Write signal pin
- <code>T6963C_ctrlrd</code>: Read signal pin
- <code>T6963C_ctrlcd</code>: Command/Data signal pin
- <code>T6963C_ctrlrst</code>: Reset signal pin
- <code>T6963C_ctrlwr_Direction</code>: Direction of Write signal pin
- <code>T6963C_ctrlrd_Direction</code>: Direction of Read signal pin
- <code>T6963C_ctrlcd_Direction</code>: Direction of Command/Data signal pin
- <code>T6963C_ctrlrst_Direction</code>: Direction of Reset signal pin

must be defined before using this function.</td></tr>
</table>

Example	```// T6963C module connections```
	```var T6963C_dataPort : byte at PORTB; // DATA port```
	
	```var T6963C_ctrlwr : sbit at LATF2_bit; // WR write signal```
	```var T6963C_ctrlrd : sbit at LATF1_bit; // RD read signal```
	```var T6963C_ctrlcd : sbit at LATF0_bit; // CD command/data signal```
	```var T6963C_ctrlrst : sbit at LATF4_bit; // RST reset signal```
	```var T6963C_ctrlwr_Direction : sbit at TRISF2_bit; // WR write signal direction```
	```var T6963C_ctrlrd_Direction : sbit at TRISF1_bit; // RD read signal direction```
	```var T6963C_ctrlcd_Direction : sbit at TRISF0_bit; // CD command/data signal direction```
	```var T6963C_ctrlrst_Direction : sbit at TRISF4_bit; // RST reset signal direction```
	
	```// Signals not used by library, they are set in main function```
	```var T6963C_ctrlce : sbit at LATF3_bit; // CE signal```
	```var T6963C_ctrlfs : sbit at LATF6_bit; // FS signal```
	```var T6963C_ctrlmd : sbit at LATF5_bit; // MD signal```
	```var T6963C_ctrlce_Direction : sbit at TRISF3_bit; // CE signal direction```
	```var T6963C_ctrlfs_Direction : sbit at TRISF6_bit; // FS signal direction```
	```var T6963C_ctrlmd_Direction : sbit at TRISF5_bit; // MD signal direction```
	```// End T6963C module connections```
	
	```...```
	```// init display for 240 pixel width, 128 pixel height and 8 bits character width```
	```T6963C_init(240, 128, 8);```

Notes	None.

T6963C_writeData

Prototype	`procedure T6963C_writeData(mydata : byte);`
Description	Writes data to T6963C controller.
Parameters	- `mydata`: data to be written
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_writeData(AddrL);`
Notes	None.

T6963C_writeCommand

Prototype	`procedure T6963C_writeCommand(mydata : byte);`
Description	Writes command to T6963C controller.
Parameters	- `mydata:` command to be written
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_writeCommand(T6963C_CURSOR_POINTER_SET);`
Notes	None.

T6963C_setPtr

Prototype	`procedure T6963C_setPtr(p : word; c : byte);`
Description	Sets the memory pointer `p` for command `p`.
Parameters	- `p`: address where command should be written - `c`: command to be written
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_setPtr(T6963C_grHomeAddr + start, T6963C_ADDRESS_POINTER_SET);`
Notes	None.

T6963C_waitReady

Prototype	`procedure T6963C_waitReady();`
Description	Pools the status byte, and loops until Toshiba Glcd module is ready.
Parameters	None.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_waitReady();`
Notes	None.

T6963C_fill

Prototype	`procedure T6963C_fill(v : byte; start, len : word);`
Description	Fills controller memory block with given byte.
Parameters	- `v`: byte to be written - `start`: starting address of the memory block - `len`: length of the memory block in bytes
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_fill(0x33,0x00FF,0x000F);`
Notes	None.

T6963C_dot

Prototype	`procedure T6963C_dot(x, y : integer; color : byte);`
Description	Draws a dot in the current graphic panel of Glcd at coordinates (x, y).
Parameters	- `x`: dot position on x-axis - `y`: dot position on y-axis - `color`: color parameter. Valid values: T6963C_BLACK and T6963C_WHITE
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_dot(x0, y0, pcolor);`
Notes	None.

T6963C_write_char

Prototype	`procedure T6963C_write_char(c, x, y, mode : byte);`
Description	Writes a char in the current text panel of Glcd at coordinates (x, y).
Parameters	- `c`: char to be written - `x`: char position on x-axis - `y`: char position on y-axis - `mode`: mode parameter. Valid values: T6963C_ROM_MODE_OR, T6963C_ROM_MODE_XOR, T6963C_ROM_MODE_AND and T6963C_ROM_MODE_TEXT Mode parameter explanation: - OR Mode: In the OR-Mode, text and graphics can be displayed and the data is logically “OR-ed”. This is the most common way of combining text and graphics for example labels on buttons. - XOR-Mode: In this mode, the text and graphics data are combined via the logical “exclusive OR”. This can be useful to display text in the negative mode, i.e. white text on black background. - AND-Mode: The text and graphic data shown on display are combined via the logical “AND function”. - TEXT-Mode: This option is only available when displaying just a text. The Text Attribute values are stored in the graphic area of display memory. For more details see the T6963C datasheet.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_write_char('A',22,23,T6963C_ROM_MODE_AND);`
Notes	None.

T6963C_write_text

Prototype	`procedure T6963C_write_text(var str : array[10] of byte; x, y, mode : byte);`
Description	Writes text in the current text panel of Glcd at coordinates (x, y).
Parameters	- `str`: text to be written - `x`: text position on x-axis - `y`: text position on y-axis - `mode`: mode parameter. Valid values: T6963C_ROM_MODE_OR, T6963C_ROM_MODE_XOR, T6963C_ROM_MODE_AND and T6963C_ROM_MODE_TEXT Mode parameter explanation: - OR Mode: In the OR-Mode, text and graphics can be displayed and the data is logically “OR-ed”. This is the most common way of combining text and graphics for example labels on buttons. - XOR-Mode: In this mode, the text and graphics data are combined via the logical “exclusive OR”. This can be useful to display text in the negative mode, i.e. white text on black background. - AND-Mode: The text and graphic data shown on display are combined via the logical “AND function”. - TEXT-Mode: This option is only available when displaying just a text. The Text Attribute values are stored in the graphic area of display memory. For more details see the T6963C datasheet.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_write_text('GLCD LIBRARY DEMO, WELCOME !', 0, 0, T6963C_ROM_MODE_XOR);`
Notes	None.

T6963C_line

Prototype	`procedure T6963C_line(x0, y0, x1, y1 : integer; pcolor : byte);`
Description	Draws a line from (x0, y0) to (x1, y1).
Parameters	- `x0`: x coordinate of the line start - `y0`: y coordinate of the line end - `x1`: x coordinate of the line start - `y1`: y coordinate of the line end - `pcolor`: color parameter. Valid values: T6963C_BLACK and T6963C_WHITE
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_line(0, 0, 239, 127, T6963C_WHITE);`
Notes	None.

T6963C_rectangle

Prototype	`procedure T6963C_rectangle(x0, y0, x1, y1 : integer; pcolor : byte);`
Description	Draws a rectangle on Glcd.
Parameters	- `x0`: x coordinate of the upper left rectangle corner - `y0`: y coordinate of the upper left rectangle corner - `x1`: x coordinate of the lower right rectangle corner - `y1`: y coordinate of the lower right rectangle corner - `pcolor`: color parameter. Valid values: T6963C_BLACK and T6963C_WHITE
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_rectangle(20, 20, 219, 107, T6963C_WHITE);`
Notes	None.

T6963C_rectangle_round_edges

Prototype	`procedure T6963C_rectangle_round_edges(x0, y0, x1, y1, radius : integer; pcolor : byte);`
Description	Draws a rounded edge rectangle on Glcd.
Parameters	- `x0`: x coordinate of the upper left rectangle corner - `y0`: y coordinate of the upper left rectangle corner - `x1`: x coordinate of the lower right rectangle corner - `y1`: y coordinate of the lower right rectangle corner - `round_radius`: radius of the rounded edge. - `pcolor`: color parameter. Valid values: `T6963C_BLACK` and `T6963C_WHITE`
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_rectangle_round_edges(20, 20, 219, 107, 12, T6963C_WHITE);`
Notes	None.

T6963C_rectangle_round_edges_fill

Prototype	`procedure T6963C_rectangle_round_edges_fill(x0, y0, x1, y1, radius : integer; pcolor : byte);`
Description	Draws a filled rounded edge rectangle on Glcd.
Parameters	- `x0`: x coordinate of the upper left rectangle corner - `y0`: y coordinate of the upper left rectangle corner - `x1`: x coordinate of the lower right rectangle corner - `y1`: y coordinate of the lower right rectangle corner - `round_radius`: radius of the rounded edge - `pcolor`: color parameter. Valid values: `T6963C_BLACK` and `T6963C_WHITE`
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_rectangle_round_edges_fill(20, 20, 219, 107, 12, T6963C_WHITE);`
Notes	None.

T6963C_box

Prototype	`procedure T6963C_box(x0, y0, x1, y1 : integer; pcolor : byte);`
Description	Draws a box on Glcd
Parameters	- `x0`: x coordinate of the upper left box corner - `y0`: y coordinate of the upper left box corner - `x1`: x coordinate of the lower right box corner - `y1`: y coordinate of the lower right box corner - `pcolor`: color parameter. Valid values: T6963C_BLACK and T6963C_WHITE
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_box(0, 119, 239, 127, T6963C_WHITE);`
Notes	None.

T6963C_circle

Prototype	`procedure T6963C_circle(x, y : integer; r : longint; pcolor : word);`
Description	Draws a circle on Glcd.
Parameters	- `x`: x coordinate of the circle center - `y`: y coordinate of the circle center - `r`: radius size - `pcolor`: color parameter. Valid values: T6963C_BLACK and T6963C_WHITE
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_circle(120, 64, 110, T6963C_WHITE);`
Notes	None.

T6963C_circle_fill

Prototype	`procedure T6963C_Circle_fill(x, y : integer; r : longint; pcolor : word);`
Description	Draws a filled circle on Glcd.
Parameters	- `x`: x coordinate of the circle center - `y`: y coordinate of the circle center - `r`: radius size - `pcolor`: color parameter. Valid values: `T6963C_BLACK` and `T6963C_WHITE`
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_circle_fill(120, 64, 110, T6963C_WHITE);`
Notes	None.

T6963C_image

Prototype	`procedure T6963C_image(const pic : ^byte);`
Description	Displays bitmap on Glcd.
Parameters	- `pic`: image to be displayed. Bitmap array can be located in both code and RAM memory (due to the mikroPascal PRO for PIC32 pointer to const and pointer to RAM equivalency).
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_image(my_image);`
Notes	Image dimension must match the display dimension. Use the integrated Glcd Bitmap Editor (menu option **Tools › Glcd Bitmap Editor**) to convert image to a constant array suitable for displaying on Glcd.

T6963C_PartialImage

Prototype	`procedure T6963C_PartialImage(x_left, y_top, width, height, picture_width, picture_height : word; const image : ^byte);`
Description	Displays a partial area of the image on a desired location.
Parameters	- `x_left`: x coordinate of the desired location (upper left coordinate). - `y_top`: y coordinate of the desired location (upper left coordinate). - `width`: desired image width. - `height`: desired image height. - `picture_width`: width of the original image. - `picture_height`: height of the original image. - `image`: image to be displayed. Bitmap array can be located in both code and RAM memory (due to the mikroPascal PRO for PIC pointer to const and pointer to RAM equivalency).
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See T6963C_init routine.
Example	`// Draws a 10x15 part of the image starting from the upper left corner on the coordinate (10,12). Original image size is 16x32.` `T6963C_PartialImage(10, 12, 10, 15, 16, 32, @image);`
Notes	Use the integrated Glcd Bitmap Editor (menu option **Tools › Glcd Bitmap Editor**) to convert image to a constant array suitable for displaying on Glcd.

T6963C_sprite

Prototype	`procedure T6963C_sprite(px, py : byte; const pic : ^byte; sx, sy : byte);`
Description	Fills graphic rectangle area (px, py) to (px+sx, py+sy) with custom size picture.
Parameters	- `px`: x coordinate of the upper left picture corner. Valid values: multiples of the font width - `py`: y coordinate of the upper left picture corner - `pic`: picture to be displayed - `sx`: picture width. Valid values: multiples of the font width - `sy`: picture height
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_sprite(76, 4, einstein, 88, 119); // draw a sprite`
Notes	If `px` and `sx` parameters are not multiples of the font width they will be scaled to the nearest lower number that is a multiple of the font width.

T6963C_set_cursor

Prototype	`procedure T6963C_set_cursor(x, y : byte);`
Description	Sets cursor to row x and column y.
Parameters	- `x`: cursor position row number - `y`: cursor position column number
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_set_cursor(cposx, cposy);`
Notes	None.

T6963C_displayGrPanel

Prototype	`procedure T6963C_displayGrPanel(n : word);`
Description	Display selected graphic panel.
Parameters	- `n`: graphic panel number. Valid values: `0` and `1`.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`// display graphic panel 1` `T6963C_displayGrPanel(1);`
Notes	None.

T6963C_displayTxtPanel

Prototype	`procedure T6963C_displayTxtPanel(n : word);`
Description	Display selected text panel.
Parameters	- `n`: text panel number. Valid values: `0` and `1`.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`// display text panel 1` `T6963C_displayTxtPanel(1);`
Notes	None.

T6963C_setGrPanel

Prototype	`procedure T6963C_setGrPanel(n : word);`
Description	Compute start address for selected graphic panel and set appropriate internal pointers. All subsequent graphic operations will be preformed at this graphic panel.
Parameters	- `n`: graphic panel number. Valid values: `0` and `1`.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`// set graphic panel 1 as current graphic panel.` `T6963C_setGrPanel(1);`
Notes	None.

T6963C_setTxtPanel

Prototype	`procedure T6963C_setTxtPanel(n : word);`
Description	Compute start address for selected text panel and set appropriate internal pointers. All subsequent text operations will be preformed at this text panel.
Parameters	- `n`: text panel number. Valid values: `0` and `1`.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`// set text panel 1 as current text panel.` `T6963C_setTxtPanel(1);`
Notes	None.

T6963C_panelFill

Prototype	`procedure T6963C_panelFill(v : word);`
Description	Fill current panel in full (graphic+text) with appropriate value (0 to clear).
Parameters	- `v`: value to fill panel with.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`clear current panel` `T6963C_panelFill(0);`
Notes	None.

T6963C_grFill

Prototype	`procedure T6963C_grFill(v: word);`
Description	Fill current graphic panel with appropriate value (0 to clear).
Parameters	- `v`: value to fill graphic panel with.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`// clear current graphic panel` `T6963C_grFill(0);`
Notes	None.

T6963C_txtFill

Prototype	`procedure T6963C_txtFill(v : word);`
Description	Fill current text panel with appropriate value (0 to clear).
Parameters	- `v`: this value increased by 32 will be used to fill text panel.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`// clear current text panel` `T6963C_txtFill(0);`
Notes	None.

T6963C_cursor_height

Prototype	`procedure T6963C_cursor_height(n: word);`
Description	Set cursor size.
Parameters	- `n`: cursor height. Valid values: `0..7`.
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_cursor_height(7);`
Notes	None.

T6963C_graphics

Prototype	`procedure T6963C_graphics(n : word);`
Description	Enable/disable graphic displaying.
Parameters	- `n`: graphic enable/disable parameter. Valid values: `0` (disable graphic dispaying) and `1` (enable graphic displaying).
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`// enable graphic displaying` `T6963C_graphics(1);`
Notes	None.

T6963C_text

Prototype	`procedure T6963C_text(n : word);`
Description	Enable/disable text displaying.
Parameters	- `n`: on/off parameter. Valid values: `0` (disable text displaying) and `1` (enable text displaying).
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`// enable text displaying` `T6963C_text(1);`
Notes	None.

T6963C_cursor

Prototype	`procedure T6963C_cursor(n : word);`
Description	Set cursor on/off.
Parameters	- `n`: on/off parameter. Valid values: `0` (set cursor off) and `1` (set cursor on).
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`// set cursor on` `T6963C_cursor(1);`
Notes	None.

T6963C_cursor_blink

Prototype	`procedure T6963C_cursor_blink(n : word);`
Description	Enable/disable cursor blinking.
Parameters	- `n`: cursor blinking enable/disable parameter. Valid values: `0` (disable cursor blinking) and `1` (enable cursor blinking).
Returns	Nothing.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`// enable cursor blinking` `T6963C_cursor_blink(1);`
Notes	None.

Library Example

The following drawing demo tests advanced routines of the T6963C Glcd library. Hardware configurations in this example are made for the LV-32MX v6 board and PIC32MX460F512L.

Copy Code To Clipboard

```
program T6963C_240x128;

uses __Lib_T6963C_Consts;

// T6963C module connections
var T6963C_dataPort : byte at PORTD;                         // DATA port
var T6963C_cntlPort : byte at PORTE;                         // DATA port

var T6963C_ctrlwr  : sbit at LATE2_bit;                      // WR write signal
var T6963C_ctrlrd  : sbit at LATE1_bit;                      // RD read signal
var T6963C_ctrlcd  : sbit at LATE0_bit;                      // CD command/data signal
var T6963C_ctrlrst : sbit at LATE4_bit;                      // RST reset signal
var T6963C_ctrlwr_Direction  : sbit at TRISE2_bit;           // WR write signal direction
var T6963C_ctrlrd_Direction  : sbit at TRISE1_bit;           // RD read signal direction
var T6963C_ctrlcd_Direction  : sbit at TRISE0_bit; // CD command/data signal direction
var T6963C_ctrlrst_Direction : sbit at TRISE4_bit;        // RST reset signal direction

// Signals not used by library, they are set in main function
var T6963C_ctrlce : sbit at LATE3_bit;                       // CE signal
var T6963C_ctrlfs : sbit at LATE6_bit;                       // FS signal
var T6963C_ctrlmd : sbit at LATE5_bit;                       // MD signal
var T6963C_ctrlce_Direction : sbit at TRISE3_bit;            // CE signal direction
var T6963C_ctrlfs_Direction : sbit at TRISE6_bit;            // FS signal direction
var T6963C_ctrlmd_Direction : sbit at TRISE5_bit;            // MD signal direction
// End T6963C module connections

var   panel : byte;         // current panel
          i : word;         // general purpose register
      curs  : byte;         // cursor visibility
      cposx,
```

```
      cposy : word;         // cursor x-y position
      txtcols : byte;       // number of text coloms
      txt, txt1 : string[29];
      txt2 : string[21];

begin
  {$DEFINE LINE_DEMO}            // Uncomment to demonstrate line drawing routines
  {$DEFINE FILL_DEMO}            // Uncomment to demonstrate fill routines
  {$DEFINE PARTIAL_IMAGE_DEMO}   // Uncomment to demonstrate partial image routine

  txt1 := ' EINSTEIN WOULD HAVE LIKED mE';
  txt  := ' GLCD LIBRARY DEMO, WELCOME !';
  txt2 := ' Partial image demo! ';

  CHECON := 0x32;
  AD1PCFG := 0xFFFF;

  TRISB8_bit := 1;               // Set RB8 as input
  TRISB9_bit := 1;               // Set RB9 as input
  TRISB10_bit := 1;              // Set RB10 as input
  TRISB11_bit := 1;              // Set RB11 as input
  TRISB12_bit := 1;              // Set RB12 as input
  TRISB13_bit := 1;              // Set RB13 as input

  T6963C_ctrlce_Direction := 0;
  T6963C_ctrlce := 0;              // Enable T6963C
  T6963C_ctrlfs_Direction := 0;
  T6963C_ctrlfs := 0;              // Font Select 8x8
  T6963C_ctrlmd_Direction := 0;
  T6963C_ctrlmd := 0;              // Column number select

  panel := 0;
  i := 0;
  curs := 0;
  cposx := 0;
  cposy := 0;

  // Initialize T6369C
  T6963C_init(240, 128, 8);

  {*
   * Enable both graphics and text display at the same time
   *}
  T6963C_graphics(1);
  T6963C_text(1);

  {*
   * Text messages
   *}
  T6963C_write_text(txt, 0, 0, T6963C_ROM_MODE_XOR);
  T6963C_write_text(txt1, 0, 15, T6963C_ROM_MODE_XOR);
```

```
{*
 * Cursor
 *}
T6963C_cursor_height(8);       // 8 pixel height
T6963C_set_cursor(0, 0);       // Move cursor to top left
T6963C_cursor(0);              // Cursor off

{*
 * Draw solid boxes
 *}
T6963C_box(0, 0, 239, 8, T6963C_WHITE);
T6963C_box(0, 119, 239, 127, T6963C_WHITE);

{*
 * Draw rectangles
 *}
{$IFDEF LINE_DEMO}
  T6963C_rectangle(0, 0, 239, 127, T6963C_WHITE);
  T6963C_rectangle(20, 20, 219, 107, T6963C_WHITE);
  T6963C_rectangle(40, 40, 199, 87, T6963C_WHITE);
  T6963C_rectangle(60, 60, 179, 67, T6963C_WHITE);
{$ENDIF}

{*
 * Draw rounded edge rectangle
 *}
{$IFDEF LINE_DEMO}
  T6963C_Rectangle_Round_Edges(10, 10, 229, 117, 12, T6963C_WHITE);
  T6963C_Rectangle_Round_Edges(30, 30, 209, 97, 12, T6963C_WHITE);
  T6963C_Rectangle_Round_Edges(50, 50, 189, 77, 12, T6963C_WHITE);
{$ENDIF}

{*
 * Draw filled rounded edge rectangle
 *}
{$IFDEF FILL_DEMO}
  T6963C_Rectangle_Round_Edges_Fill(10, 10, 229, 117, 12, T6963C_WHITE);
  T6963C_Rectangle_Round_Edges_Fill(20, 20, 219, 107, 12, T6963C_BLACK);
  T6963C_Rectangle_Round_Edges_Fill(30, 30, 209, 97, 12, T6963C_WHITE);
  T6963C_Rectangle_Round_Edges_Fill(40, 40, 199, 87, 12, T6963C_BLACK);
  T6963C_Rectangle_Round_Edges_Fill(50, 50, 189, 77, 12, T6963C_WHITE);
{$ENDIF}

{*
 * Draw a cross
 *}
{$IFDEF LINE_DEMO}
  T6963C_line(0, 0, 239, 127, T6963C_WHITE);
  T6963C_line(0, 127, 239, 0, T6963C_WHITE);
{$ENDIF}
```

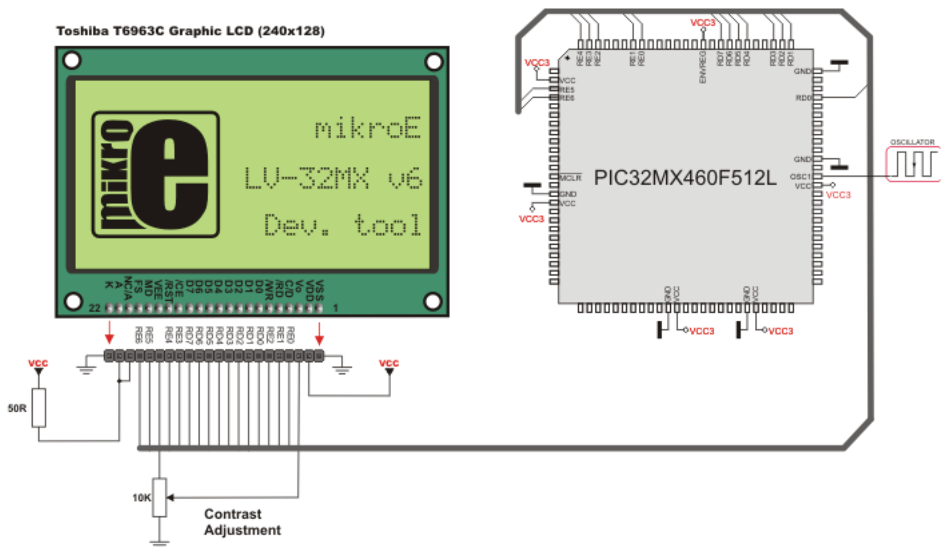
```
  {*
   * Draw circles
   *}
  {$IFDEF LINE_DEMO}
    T6963C_circle(120, 64, 10, T6963C_WHITE);
    T6963C_circle(120, 64, 30, T6963C_WHITE);
    T6963C_circle(120, 64, 50, T6963C_WHITE);
    T6963C_circle(120, 64, 70, T6963C_WHITE);
    T6963C_circle(120, 64, 90, T6963C_WHITE);
    T6963C_circle(120, 64, 110, T6963C_WHITE);
    T6963C_circle(120, 64, 130, T6963C_WHITE);
   {$ENDIF}

  {*
   * Draw filled circles
   *}
  {$IFDEF FILL_DEMO}
    T6963C_circle_fill(120, 64, 60, T6963C_WHITE);
    T6963C_circle_fill(120, 64, 55, T6963C_BLACK);
    T6963C_circle_fill(120, 64, 50, T6963C_WHITE);
    T6963C_circle_fill(120, 64, 45, T6963C_BLACK);
    T6963C_circle_fill(120, 64, 40, T6963C_WHITE);
    T6963C_circle_fill(120, 64, 35, T6963C_BLACK);
    T6963C_circle_fill(120, 64, 30, T6963C_WHITE);
    T6963C_circle_fill(120, 64, 25, T6963C_BLACK);
    T6963C_circle_fill(120, 64, 20, T6963C_WHITE);
    T6963C_circle_fill(120, 64, 15, T6963C_BLACK);
    T6963C_circle_fill(120, 64, 10, T6963C_WHITE);
    T6963C_circle_fill(120, 64, 5, T6963C_BLACK);
  {$ENDIF}

  Delay_ms(1000);
  T6963C_sprite(76, 4, @einstein, 88, 119);      // Draw a sprite
  Delay_ms(1000);

  T6963C_setGrPanel(1);                          // Select other graphic panel

  T6963C_image(@mikroE_240x128_bmp);
  T6963C_displayGrPanel(1);
  Delay_ms(1000);
  {$IFDEF PARTIAL_IMAGE_DEMO}
    T6963C_grFill(0);
   T6963C_PartialImage(0, 0, 64, 64, 240, 128, @mikroE_240x128_bmp);  // Display partial
image
    Delay_ms(1000);
  T6963C_graphics(0);
  {$ENDIF}
  T6963C_image(@mikroE_240x128_bmp);
  T6963C_graphics(1);
  T6963C_displayGrPanel(0);
```

```
  while (TRUE) do                                // Endless loop
    begin

      {*
       * If RB8 is pressed, toggle the display between graphic panel 0 and graphic 1
       *}
      if(RB8_bit <> 0) then
        begin
          Inc(panel) ;
          panel := panel and 1;
          T6963C_setPtr((T6963C_grMemSize + T6963C_txtMemSize) * panel, T6963C_GRAPHIC_
HOME_ADDRESS_SET) ;
          Delay_ms(300) ;
        end

      {*
       * If RB9 is pressed, display only graphic panel
       *}
      else
        if (RB9_bit <> 0) then
          begin
            T6963C_graphics(1) ;
            T6963C_text(0) ;
            Delay_ms(300) ;
          end

      {*
       * If RB10 is pressed, display only text panel
       *}
        else
          if (RB10_bit <> 0) then
            begin
              T6963C_graphics(0);
              T6963C_text(1);
              Delay_ms(300);
            end

      {*
       * If RB11 is pressed, display text and graphic panels
       *}
          else
            if (RB11_bit <> 0) then
              begin
                T6963C_graphics(1);
                T6963C_text(1);
                Delay_ms(300);
              end

      {*
       * If RB12 is pressed, change cursor
       *}
```

```
          else
           if(RB12_bit <> 0) then
             begin
               Inc(curs);
               if (curs = 3) then
                 curs := 0;
               case curs of
                 0:
                    // no cursor
                    T6963C_cursor(0);

                 1: begin
                      // blinking cursor
                      T6963C_cursor(1);
                      T6963C_cursor_blink(1);
                    end;
                 2: begin
                      // non blinking cursor
                      T6963C_cursor(1);
                      T6963C_cursor_blink(0);
                    end;
               end;
               Delay_ms(300);
             end

    {*
     * If RB13 is pressed, perform the "Partial image" demonstration
     *}
    else
      {$IFDEF PARTIAL_IMAGE_DEMO}
       if( RB13_bit = 1) then
         begin
           T6963C_setGrPanel(0);
           T6963C_setTxtPanel(0);
           T6963C_txtFill(0);
           T6963C_setGrPanel(1);
           T6963C_setTxtPanel(0);
           T6963C_graphics(1);
           T6963C_text(1);
           T6963C_displayGrPanel(1);
           T6963C_write_text(txt2, 5, 15, T6963C_ROM_MODE_XOR);
           Delay_1sec();

           T6963C_grFill(0);
           T6963C_PartialImage(0, 0, 64, 64, 240, 128, @mikroE_240x128_bmp);
           Delay_ms(1500);

           T6963C_PartialImage(0, 0, 128, 128, 240, 128, @mikroE_240x128_bmp);
           Delay_ms(1500);

           T6963C_PartialImage(0, 0, 240, 128, 240, 128, @mikroE_240x128_bmp);
           Delay_ms(1500);
```

```
            T6963C_txtFill(0);
            T6963C_write_text(txt, 0, 0, T6963C_ROM_MODE_XOR);
            T6963C_write_text(txt1, 0, 15, T6963C_ROM_MODE_XOR);
          end;
       {$ENDIF}
      {*
       * Move cursor, even if not visible
       *}
      Inc(cposx);
      if (cposx = T6963C_txtCols) then
        begin
          cposx := 0;
          Inc(cposy);
          if (cposy = T6963C_grHeight div T6963C_CHARACTER_HEIGHT) then
            cposy := 0;
        end;
      T6963C_set_cursor(cposx, cposy);

      Delay_ms(100);
    end;
end.
```

HW Connection



SPI T6963C Glcd HW connection

TFT Library

Thin film transistor liquid crystal display (TFT-LCD) is a variant of liquid crystal display (LCD) which uses thin-film transistor (TFT) technology to improve image quality (e.g., addressability, contrast).
TFT LCD is one type of active matrix LCD, though all LCD-screens are based on TFT active matrix addressing.
TFT LCDs are used in television sets, computer monitors, mobile phones, handheld video game systems, personal digital assistants, navigation systems, projectors, etc.

The mikroPascal PRO for PIC32 provides a library for working with HX8347-D 320x240 TFT Lcd controller.The HX8347-D is designed to provide a single-chip solution that combines a gate driver, a source driver, power supply circuit for 262,144 colors to drive a TFT panel with 320x240 dots at maximum.

The HX8347-D is suitable for any small portable battery-driven and long-term driving products, such as small PDAs, digital cellular phones and bi-directional pagers.

External dependencies of TFT Library

The following variables must be defined in all projects using TFT library:	Description:	Example:
`var TFT_DataPort : byte; external; sfr;`	TFT Data Port.	`var TFT_DataPort : byte at LATE;`
`var TFT_DataPort_Direction : byte; external; sfr;`	Direction of the TFT Data Port.	`var TFT_DataPort_Direction : byte at TRISE;`
`var TFT_WR : sbit; sfr; external;`	Write signal.	`var TFT_WR : sbit at LATD13_bit;`
`var TFT_RD : sbit; sfr; external;`	Read signal.	`var TFT_RD : sbit at LATD12_bit;`
`var TFT_CS : sbit; sfr; external;`	Chip Select signal.	`var TFT_CS : sbit at LATC3_bit;`
`var TFT_RS : sbit; sfr; external;`	Command/Register Select signal.	`var TFT_RS : sbit at LATB15_bit;`
`var TFT_RST : sbit; sfr; external;`	Reset signal.	`var TFT_RST : sbit at LATC1_bit;`
`var TFT_WR_Direction : sbit; sfr; external;`	Direction of the Write pin.	`var TFT_WR_Direction : sbit at TRISD13_bit;`
`var TFT_RD_Direction : sbit; sfr; external;`	Direction of the Read pin.	`var TFT_RD_Direction : sbit at TRISD12_bit;`
`var TFT_CS_Direction : sbit; sfr; external;`	Direction of the Chip Select pin.	`var TFT_CS_Direction : sbit at TRISC3_bit;`
`var TFT_RS_Direction : sbit; sfr; external;`	Direction of the Register Select pin.	`var TFT_RS_Direction : sbit at TRISB13_bit;`
`var TFT_RST_Direction : sbit; sfr; external;`	Direction of the Reset pin.	`var TFT_RST_Direction : sbit at TRISC1_bit;`

Library Routines

- TFT_Init
- TFT_Set_Index
- TFT_Write_Command
- TFT_Write_Data
- TFT_Set_Active
- TFT_Set_Font
- TFT_Write_Char
- TFT_Write_Text
- TFT_Fill_Screen
- TFT_Set_Pen
- TFT_Set_Brush
- TFT_Dot
- TFT_Line
- TFT_H_Line
- TFT_V_Line
- TFT_Rectangle
- TFT_Rectangle_Round_Edges
- TFT_Circle
- TFT_Image
- TFT_PartialImage
- TFT_Image_Jpeg
- TFT_RGBToColor16bit
- TFT_Color16bitToRGB

TFT_Init

Prototype	`procedure TFT_Init(display_width, display_height : word) ;`
Returns	Nothing
Description	Initializes TFT display in the 8-bit working mode. Parameters: - `width`: width of the TFT panel - `height`: height of the TFT panel
Requires	Global variables : - `TFT_DataPort`: Data Port - `TFT_WR`: Write signal pin - `TFT_RD`: Read signal pin - `TFT_CS`: Chip Select signal pin - `TFT_RS`: Register Select signal pin - `TFT_RST`: Reset signal pin - `TFT_DataPort_Direction`: Direction of Data Port - `TFT_WR_Direction`: Direction of Write signal pin - `TFT_RD_Direction`: Direction of Read signal pin - `TFT_CS_Direction`: Direction of Chip Select signal pin - `TFT_RS_Direction`: Direction of Register Select signal pin - `TFT_RST_Direction`: Direction of Reset signal pin must be defined before using this function.

Example	`// TFT display connections`
	`var TFT_DataPort : byte at LATE;`
	`var TFT_WR : sbit at LATD13_bit;`
	`var TFT_RD : sbit at LATD12_bit;`
	`var TFT_CS : sbit at LATC3_bit;`
	`var TFT_RS : sbit at LATB15_bit;`
	`var TFT_RST : sbit at LATC1_bit;`
	
	`var TFT_DataPort_Direction : byte at TRISE;`
	`var TFT_WR_Direction : sbit at TRISD13_bit;`
	`var TFT_RD_Direction : sbit at TRISD12_bit;`
	`var TFT_CS_Direction : sbit at TRISC3_bit;`
	`var TFT_RS_Direction : sbit at TRISB15_bit;`
	`var TFT_RST_Direction : sbit at TRISC1_bit;`
	`// End of TFT display connections`
	
	`// Initialize 240x320 TFT display`
	`TFT_Init(240, 320);`

TFT_Set_Index

Prototype	`procedure TFT_Set_Index(index : byte);`
Returns	Nothing
Description	Accesses register space of the controller and sets the desired register. Parameters: - `index`: desired register number.
Requires	TFT module needs to be initialized. See the TFT_Init routine.
Example	`// Access register at the location 0x02` `TFT_Set_Index(0x02);`

TFT_Write_Command

Prototype	`procedure TFT_Write_Command(cmd : byte);`
Returns	Nothing
Description	Accesses data space and writes a command. Parameters: - `cmd`: command to be written.
Requires	TFT module needs to be initialized. See the TFT_Init routine.
Example	`// Write a command` `TFT_Write_Command(0x02);`

TFT_Write_Data

Prototype	`procedure TFT_Write_Data(_data : word);`
Returns	Nothing
Description	Writes date into display memory. Parameters: - `_data`:data to be written.
Requires	TFT module needs to be initialized. See the TFT_Init routine.
Example	`// Send data` `TFT_Write_Data(0x02);`

TFT_Set_Active

Prototype	`procedure TFT_Set_Active(Set_Index_Ptr : ^TTFT_Set_Index_Ptr; Write_Command_Ptr : ^TTFT_Write_Command_Ptr; Write_Data_Ptr : ^TTFT_Write_Data_Ptr);`
Returns	Nothing
Description	This function sets appropriate pointers to a user-defined basic routines in order to enable multiple working modes. Parameters: - `Set_Index_Ptr`: Set_Index handler. - `Write_Command_Ptr`: _Write_Command handler. - `Write_Data_Ptr`: Write_Data handler.
Requires	None.
Example	(see code below)

```
// Example of establishing 16-bit communication between TFT display and
PORTD, PORTE of MCU :

procedure Set_Index(index : byte) {
  TFT_RS = 0;
  Lo(LATD) = index;
  TFT_WR = 0;
  TFT_WR = 1;
}

procedure Write_Command(cmd : byte) {
  TFT_RS = 1;
  Lo(LATD) = cmd;
  TFT_WR = 0;
  TFT_WR = 1;
}

procedure Write_Data(_data : word) {
  TFT_RS = 1;
  Lo(LATE) = Hi(_data);
  Lo(LATD) = Lo(_data);
  TFT_WR = 0;
  TFT_WR = 1;
}

procedure main() {
  TRISE = 0;
  TRISD = 0;

  TFT_Set_Active(Set_Index,Write_Command,Write_Data);
  TFT_Init(320, 240);

  .....
}
```

TFT_Set_Font

Prototype	`procedure TFT_Set_Font(activeFont : ^const far byte; font_color : word; font_orientation : byte);`
Returns	Nothing
Description	Sets font, its color and font orientation. Parameters: - `activeFont`: desired font. Currently, only `TFT_defaultFont` (Tahoma14x16) is supported. - `font_color`: sets font color: **Value** / **Description**: `CL_AQUA` – Aqua color `CL_BLACK` – Black color `CL_BLUE` – Blue color `CL_FUCHSIA` – Fuchsia color `CL_GRAY` – Gray color `CL_GREEN` – Green color `CL_LIME` – Lime color `CL_MAROON` – Maroon color `CL_NAVY` – Navy color `CL_OLIVE` – Olive color `CL_PURPLE` – Purple color `CL_RED` – Red color `CL_SILVER` – Silver color `CL_TEAL` – Teal color `CL_WHITE` – White color `CL_YELLOW` – Yellow color - `font_orientation`: sets font orientation: **Value** / **Description**: `FO_HORIZONTAL` – Horizontal orientation `FO_VERTICAL` – Vertical orientation
Requires	TFT module needs to be initialized. See the TFT_Init routine.
Example	`TFT_Set_Font(@TFT_defaultFont, CL_BLACK, FO_HORIZONTAL);`

Nested tables from the Description cell:

Value	Description
CL_AQUA	Aqua color
CL_BLACK	Black color
CL_BLUE	Blue color
CL_FUCHSIA	Fuchsia color
CL_GRAY	Gray color
CL_GREEN	Green color
CL_LIME	Lime color
CL_MAROON	Maroon color
CL_NAVY	Navy color
CL_OLIVE	Olive color
CL_PURPLE	Purple color
CL_RED	Red color
CL_SILVER	Silver color
CL_TEAL	Teal color
CL_WHITE	White color
CL_YELLOW	Yellow color

Value	Description
FO_HORIZONTAL	Horizontal orientation
FO_VERTICAL	Vertical orientation

TFT_Write_Char

Prototype	`procedure TFT_Write_Char(ch, x, y : word);`
Returns	Nothing.
Description	Writes a char on the TFT at coordinates (x, y). - `c`: char to be written. - `x`: char position on x-axis. - `y`: char position on y-axis.
Requires	TFT module needs to be initialized. See the TFT_Init routine.
Example	`TFT_Write_Char('A',22,23,);`

TFT_Write_Text

Prototype	`procedure TFT_Write_Text(var text : string; x, y : word);`
Returns	Nothing.
Description	Writes text on the TFT at coordinates (x, y). Parameters: - `text`: text to be written. - `x`: text position on x-axis. - `y`: text position on y-axis.
Requires	TFT module needs to be initialized. See the TFT_Init routine.
Example	`TFT_Write_Text('TFT LIBRARY DEMO, WELCOME !', 0, 0);`

TFT_Fill_Screen

<table>
<tr><td>Prototype</td><td><code>procedure TFT_Fill_Screen(color : word);</code></td></tr>
<tr><td>Returns</td><td>Nothing.</td></tr>
<tr><td>Description</td><td>Fills screen memory block with given color.

Parameters:

- <code>color</code>: color to be filled:
<table>
<tr><th>Value</th><th>Description</th></tr>
<tr><td><code>CL_AQUA</code></td><td>Aqua color</td></tr>
<tr><td><code>CL_BLACK</code></td><td>Black color</td></tr>
<tr><td><code>CL_BLUE</code></td><td>Blue color</td></tr>
<tr><td><code>CL_FUCHSIA</code></td><td>Fuchsia color</td></tr>
<tr><td><code>CL_GRAY</code></td><td>Gray color</td></tr>
<tr><td><code>CL_GREEN</code></td><td>Green color</td></tr>
<tr><td><code>CL_LIME</code></td><td>Lime color</td></tr>
<tr><td><code>CL_MAROON</code></td><td>Maroon color</td></tr>
<tr><td><code>CL_NAVY</code></td><td>Navy color</td></tr>
<tr><td><code>CL_OLIVE</code></td><td>Olive color</td></tr>
<tr><td><code>CL_PURPLE</code></td><td>Purple color</td></tr>
<tr><td><code>CL_RED</code></td><td>Red color</td></tr>
<tr><td><code>CL_SILVER</code></td><td>Silver color</td></tr>
<tr><td><code>CL_TEAL</code></td><td>Teal color</td></tr>
<tr><td><code>CL_WHITE</code></td><td>White color</td></tr>
<tr><td><code>CL_YELLOW</code></td><td>Yellow color</td></tr>
</table>
</td></tr>
<tr><td>Requires</td><td>TFT module needs to be initialized. See the TFT_Init routine.</td></tr>
<tr><td>Example</td><td><code>TFT_Fill_Screen(CL_BLACK);</code></td></tr>
</table>

TFT_Dot

Prototype	`procedure TFT_Dot(x, y : integer; color : word);`
Returns	Nothing.
Description	Draws a dot on the TFT at coordinates (x, y). Parameters: - `x`: dot position on x-axis. - `y`: dot position on y-axis. - `color`: color parameter. Valid values: **Value** — **Description** `CL_AQUA` — Aqua color `CL_BLACK` — Black color `CL_BLUE` — Blue color `CL_FUCHSIA` — Fuchsia color `CL_GRAY` — Gray color `CL_GREEN` — Green color `CL_LIME` — Lime color `CL_MAROON` — Maroon color `CL_NAVY` — Navy color `CL_OLIVE` — Olive color `CL_PURPLE` — Purple color `CL_RED` — Red color `CL_SILVER` — Silver color `CL_TEAL` — Teal color `CL_WHITE` — White color `CL_YELLOW` — Yellow color
Requires	TFT module needs to be initialized. See the TFT_Init routine.
Example	`TFT_Dot(50, 50, CL_BLACK);`

TFT_Set_Pen

<table>
<tr><td>Prototype</td><td><code>procedure TFT_Set_Pen(pen_color : word; pen_width : byte);</code></td></tr>
<tr><td>Returns</td><td>Nothing.</td></tr>
<tr><td>Description</td><td>Sets color and thickness parameter for drawing line, circle and rectangle elements.

Parameters:

- <code>pen_color</code>: Sets color.

<table>
<tr><th>Value</th><th>Description</th></tr>
<tr><td><code>CL_AQUA</code></td><td>Aqua color</td></tr>
<tr><td><code>CL_BLACK</code></td><td>Black color</td></tr>
<tr><td><code>CL_BLUE</code></td><td>Blue color</td></tr>
<tr><td><code>CL_FUCHSIA</code></td><td>Fuchsia color</td></tr>
<tr><td><code>CL_GRAY</code></td><td>Gray color</td></tr>
<tr><td><code>CL_GREEN</code></td><td>Green color</td></tr>
<tr><td><code>CL_LIME</code></td><td>Lime color</td></tr>
<tr><td><code>CL_MAROON</code></td><td>Maroon color</td></tr>
<tr><td><code>CL_NAVY</code></td><td>Navy color</td></tr>
<tr><td><code>CL_OLIVE</code></td><td>Olive color</td></tr>
<tr><td><code>CL_PURPLE</code></td><td>Purple color</td></tr>
<tr><td><code>CL_RED</code></td><td>Red color</td></tr>
<tr><td><code>CL_SILVER</code></td><td>Silver color</td></tr>
<tr><td><code>CL_TEAL</code></td><td>Teal color</td></tr>
<tr><td><code>CL_WHITE</code></td><td>White color</td></tr>
<tr><td><code>CL_YELLOW</code></td><td>Yellow color</td></tr>
</table>

- <code>pen_width</code>: sets thickness.</td></tr>
<tr><td>Requires</td><td>TFT module needs to be initialized. See the TFT_Init routine.</td></tr>
<tr><td>Example</td><td><code>TFT_Set_Pen(CL_BLACK, 10);</code></td></tr>
</table>

TFT_Set_Brush

<table>
<tr><td>Prototype</td><td><code>procedure TFT_Set_Brush(brush_enabled : byte; brush_color : word; gradient_enabled, gradient_orientation : byte; gradient_color_from, gradient_color_to : word);</code></td></tr>
<tr><td>Returns</td><td>Nothing.</td></tr>
<tr><td>Description</td><td>Sets color and gradient which will be used to fill circles or rectangles.

Parameters:

- <code>brush_enabled</code>: enable brush fill.
<table>
<tr><th>Value</th><th>Description</th></tr>
<tr><td><code>1</code></td><td>Enable brush fill.</td></tr>
<tr><td><code>0</code></td><td>Disable brush fill.</td></tr>
</table>

- <code>brush_color</code>: set brush fill color.
<table>
<tr><th>Value</th><th>Description</th></tr>
<tr><td><code>CL_AQUA</code></td><td>Aqua color</td></tr>
<tr><td><code>CL_BLACK</code></td><td>Black color</td></tr>
<tr><td><code>CL_BLUE</code></td><td>Blue color</td></tr>
<tr><td><code>CL_FUCHSIA</code></td><td>Fuchsia color</td></tr>
<tr><td><code>CL_GRAY</code></td><td>Gray color</td></tr>
<tr><td><code>CL_GREEN</code></td><td>Green color</td></tr>
<tr><td><code>CL_LIME</code></td><td>Lime color</td></tr>
<tr><td><code>CL_MAROON</code></td><td>Maroon color</td></tr>
<tr><td><code>CL_NAVY</code></td><td>Navy color</td></tr>
<tr><td><code>CL_OLIVE</code></td><td>Olive color</td></tr>
<tr><td><code>CL_PURPLE</code></td><td>Purple color</td></tr>
<tr><td><code>CL_RED</code></td><td>Red color</td></tr>
<tr><td><code>CL_SILVER</code></td><td>Silver color</td></tr>
<tr><td><code>CL_TEAL</code></td><td>Teal color</td></tr>
<tr><td><code>CL_WHITE</code></td><td>White color</td></tr>
<tr><td><code>CL_YELLOW</code></td><td>Yellow color</td></tr>
</table>
</td></tr>
</table>

Description	
	- gradient_enabled: enable gradient

- gradient_enabled: enable gradient

Value	Description
1	Enable gradient.
0	Disable gradient.

- gradient_orientation: sets gradient orientation :

Value	Description
LEFT_TO_RIGHT	Left to right gradient orientation
TOP_TO_BOTTOM	Top to bottom gradient orientation

- gradient_color_from: sets the starting gradient color.

Value	Description
CL_AQUA	Aqua color
CL_BLACK	Black color
CL_BLUE	Blue color
CL_FUCHSIA	Fuchsia color
CL_GRAY	Gray color
CL_GREEN	Green color
CL_LIME	Lime color
CL_MAROON	Maroon color
CL_NAVY	Navy color
CL_OLIVE	Olive color
CL_PURPLE	Purple color
CL_RED	Red color
CL_SILVER	Silver color
CL_TEAL	Teal color
CL_WHITE	White color
CL_YELLOW	Yellow color

<table>
<tr><td>Description</td><td>- <code>gradient_color_to</code>: sets the ending gradient color.
<table>
<tr><th>Value</th><th>Description</th></tr>
<tr><td><code>CL_AQUA</code></td><td>Aqua color</td></tr>
<tr><td><code>CL_BLACK</code></td><td>Black color</td></tr>
<tr><td><code>CL_BLUE</code></td><td>Blue color</td></tr>
<tr><td><code>CL_FUCHSIA</code></td><td>Fuchsia color</td></tr>
<tr><td><code>CL_GRAY</code></td><td>Gray color</td></tr>
<tr><td><code>CL_GREEN</code></td><td>Green color</td></tr>
<tr><td><code>CL_LIME</code></td><td>Lime color</td></tr>
<tr><td><code>CL_MAROON</code></td><td>Maroon color</td></tr>
<tr><td><code>CL_NAVY</code></td><td>Navy color</td></tr>
<tr><td><code>CL_OLIVE</code></td><td>Olive color</td></tr>
<tr><td><code>CL_PURPLE</code></td><td>Purple color</td></tr>
<tr><td><code>CL_RED</code></td><td>Red color</td></tr>
<tr><td><code>CL_SILVER</code></td><td>Silver color</td></tr>
<tr><td><code>CL_TEAL</code></td><td>Teal color</td></tr>
<tr><td><code>CL_WHITE</code></td><td>White color</td></tr>
<tr><td><code>CL_YELLOW</code></td><td>Yellow color</td></tr>
</table>
</td></tr>
<tr><td>Requires</td><td>TFT module needs to be initialized. See the TFT_Init routine.</td></tr>
<tr><td>Example</td><td><code>// Enable gradient from black to white color, left-right orientation</code>
<code>TFT_Set_Brush(0, 0, 1, LEFT_TO_RIGHT, CL_BLACK, CL_WHITE);</code></td></tr>
</table>

TFT_Line

<table>
<tr><td>Prototype</td><td><code>procedure TFT_Line(x1, y1, x2, y2 : integer);</code></td></tr>
<tr><td>Returns</td><td>Nothing.</td></tr>
<tr><td>Description</td><td>Draws a line from (x1, y1) to (x2, y2).

Parameters:

- <code>x1</code>: x coordinate of the line start.
- <code>y1</code>: y coordinate of the line end.
- <code>x2</code>: x coordinate of the line start.
- <code>y2</code>: y coordinate of the line end.</td></tr>
<tr><td>Requires</td><td>TFT module needs to be initialized. See the TFT_Init routine.</td></tr>
<tr><td>Example</td><td><code>TFT_Line(0, 0, 239, 127);</code></td></tr>
</table>

TFT_H_Line

Prototype	`procedure TFT_H_Line(x_start, x_end, y_pos : integer);`
Returns	Nothing.
Description	Draws a horizontal line on TFT. Parameters: - `x_start`: x coordinate of the line start. - `x_end`: x coordinate of the line end. - `y_pos`: y coordinate of horizontal line.
Requires	TFT module needs to be initialized. See the TFT_Init routine.
Example	`// Draw a horizontal line between dots (10,20) and (50,20)` `TFT_H_Line(10, 50, 20);`

TFT_V_Line

Prototype	`procedure TFT_V_Line(y_start, y_end, x_pos : integer);`
Returns	Nothing.
Description	Draws a vertical line on TFT. Parameters: - `y_start`: y coordinate of the line start. - `y_end`: y coordinate of the line end. - `x_pos`: x coordinate of vertical line.
Requires	TFT module needs to be initialized. See the TFT_Init routine.
Example	`// Draw a vertical line between dots (10,5) and (10,25)` `TFT_V_Line(5, 25, 10);`

TFT_Rectangle

Prototype	`procedure TFT_Rectangle(x_upper_left, y_upper_left, x_bottom_right, y_bottom_right:integer);`
Returns	Nothing.
Description	Draws a rectangle on TFT. Parameters: - `x_upper_left`: x coordinate of the upper left rectangle corner. - `y_upper_left`: y coordinate of the upper left rectangle corner. - `x_bottom_right`: x coordinate of the lower right rectangle corner. - `y_bottom_right`: y coordinate of the lower right rectangle corner.
Requires	TFT module needs to be initialized. See the TFT_Init routine.
Example	`TFT_Rectangle(20, 20, 219, 107);`

TFT_Rectangle_Round_Edges

Prototype	`procedure TFT_Rectangle_Round_Edges(x_upper_left, y_upper_left, x_bottom_right, y_bottom_right, round_radius : word);`
Returns	Nothing.
Description	Draws a rounded edge rectangle on TFT. Parameters: - `x_upper_left`: x coordinate of the upper left rectangle corner. - `y_upper_left`: y coordinate of the upper left rectangle corner. - `x_bottom_right`: x coordinate of the lower right rectangle corner. - `y_bottom_right`: y coordinate of the lower right rectangle corner. - `round_radius`: radius of the rounded edge.
Requires	TFT module needs to be initialized. See the TFT_Init routine.
Example	`TFT_Rectangle_Round_Edges(20, 20, 219, 107, 12);`

TFT_Circle

Prototype	`procedure TFT_Circle(x_center, y_center, radius : integer);`
Returns	Nothing.
Description	Draws a circle on TFT. Parameters: - `x`: x coordinate of the circle center. - `y`: y coordinate of the circle center. - `r`: radius size.
Requires	TFT module needs to be initialized. See the TFT_Init routine.
Example	`TFT_Circle(120, 64, 110);`

TFT_Image

Prototype	`procedure TFT_Image(left, top : word; image : ^const far byte; stretch : byte);`
Returns	Nothing.
Description	Displays an image on a desired location. Parameters: - `left`: position of the image's left edge. - `top`:position of the image's top edge. - `image`: image to be displayed. Bitmap array is located in code memory. - `stretch`: stretches image by a given factor (if 2, it will double the image.).
Requires	TFT module needs to be initialized. See the TFT_Init routine.
Example	`TFT_Image(0, 0, image, 1);`

TFT_Partial_Image

Prototype	`procedure TFT_Partial_Image(left, top, width, height : word; image : ^const far byte; stretch : byte) ;`
Returns	Nothing.
Description	Displays a partial area of the image on a desired location. Parameters: - `left`: left coordinate of the image. - `top`: top coordinate of the image. - `width`: desired image width. - `height`: desired image height. - `image`: image to be displayed. Bitmap array is located in code memory. - `stretch`: stretches image by a given factor (if 2, it will double the image.).
Requires	TFT module needs to be initialized. See the TFT_Init routine.
Example	`// Draws a 10x15 part of the image starting from the upper left corner on the coordinate (10,12)` `TFT_PartialImage(10, 12, 10, 15, @image, 1);`

TFT_Image_Jpeg

Prototype	`function TFT_Image_Jpeg(left, top : word; image : ^const far byte): byte;`
Returns	- `0` - if image is loaded and displayed successfully. - `1` - if error occured.
Description	Displays a JPEG image on a desired location. Parameters: - `left`: left coordinate of the image. - `top`: top coordinate of the image. - `image`: image to be displayed. Bitmap array is located in code memory.
Requires	TFT module needs to be initialized. See the TFT_Init routine.
Example	`TFT_Image_Jpeg(0, 0, image);`

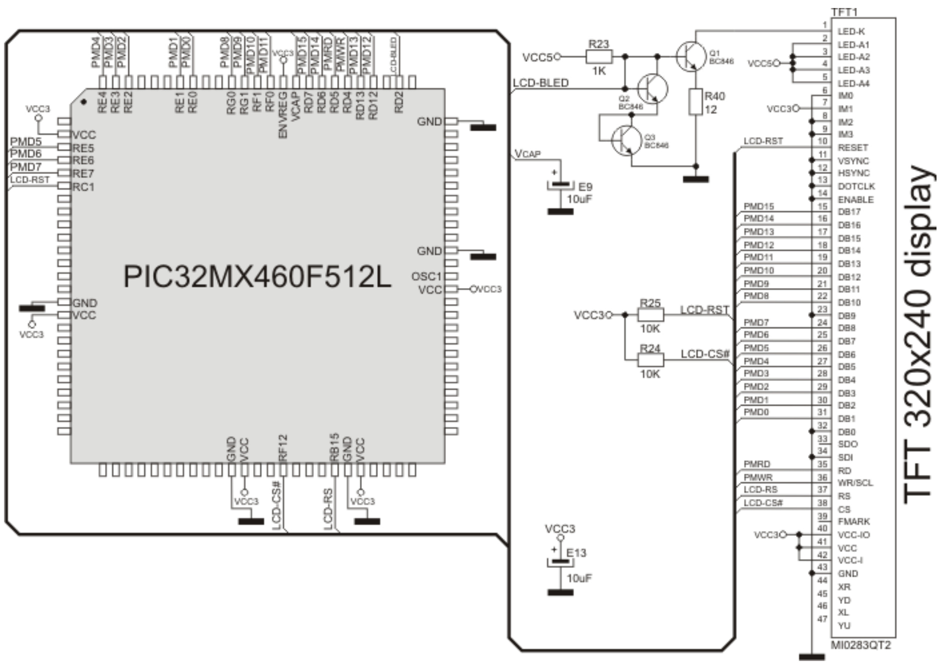
TFT_RGBToColor16bit

Prototype	`function TFT_RGBToColor16bit(rgb_red, rgb_green, rgb_blue : byte) : word;`
Returns	Returns a color value in the following bit-order : 5 bits red, 6 bits green and 5 bits blue color.
Description	Converts 5:6:5 RGB format into true color format. Parameters: - `rgb_red`: red component of the image. - `rgb_green`: green component of the image. - `rgb_blue`: blue component of the image.
Requires	TFT module needs to be initialized. See the TFT_Init routine.
Example	`color16 = TFT_Image_Jpeg(150, 193, 65);`

TFT_Color16bitToRGB

Prototype	`procedure TFT_Color16bitToRGB(color : word; rgb_red, rgb_green, rgb_blue : ^byte);`
Returns	Nothing.
Description	Converts true color into 5:6:5 RGB format. Parameters: - `color`: true color to be converted. - `rgb_red`: red component of the input color. - `rgb_green`: green component of the input color. - `rgb_blue`: blue component of the input color.
Requires	TFT module needs to be initialized. See the TFT_Init routine.
Example	`TFT_Color16bitToRGB(start_color, @red_start, @green_start, @blue_start);`

HW Connection



TFT HW connection

Touch Panel Library

The mikroPascal PRO for PIC32 provides a library for working with Touch Panel.

Library Dependency Tree

External dependencies of Touch Panel Library

The following variables must be defined in all projects using Touch Panel Library:	Description:	Example:
`var DriveA : sbit; sfr; external;`	DriveA line.	`var DriveA : sbit at LATC13_bit;`
`var DriveB : sbit; sfr; external;`	DriveB line.	`var DriveB : sbit at LATC14_bit;`
`var DriveA_Direction : sbit; sfr; external;`	Direction of the DriveA pin.	`var DriveA_Direction : sbit at TRISC13_bit;`
`var DriveB_Direction : sbit; sfr; external;`	Direction of the DriveB pin.	`var DriveB_Direction : sbit at TRISC14_bit;`

Library Routines

- TP_Init
- TP_Set_ADC_Threshold
- TP_Press_Detect
- TP_Get_Coordinates
- TP_Calibrate_Bottom_Left
- TP_Calibrate_Upper_Right
- TP_Get_Calibration_Consts
- TP_Set_Calibration_Consts

TP_Init

Prototype	`procedure TP_Init(display_width : word; display_height : word; readX_ChNo : byte; readY_ChNo : byte);`
Description	Initialize touch panel display. Default touch panel ADC threshold value is set to 3900.
Parameters	- `display_width:` set display width. - `display_height:` set display height. - `readX_ChNo:` read X coordinate from desired ADC channel. - `readY_ChNo:` read Y coordinate from desired ADC channel.
Returns	Nothing.
Requires	Before calling this function initialize ADC module.

Example	`ADC1_Init();                  // Initalize ADC module`
	`TP_Init(128, 64, 6, 7);     // Initialize touch panel, dimensions 128x64`

Notes	None.

TP_Set_ADC_Threshold

Prototype	`procedure TP_Set_ADC_Threshold(threshold : word);`
Description	Set custom ADC threshold value, call this function after TP_Init.
Parameters	- `threshold`: custom ADC threshold value.
Returns	Nothing.
Requires	TP_Init has to be called before using this routine.
Example	`TP_Set_ADC_Threshold(3900);    // Set touch panel ADC threshold`
Notes	None.

TP_Press_Detect

Prototype	`function TP_Press_Detect() : byte;`
Description	Detects if the touch panel has been pressed.
Parameters	None.
Returns	- `1` - if touch panel is pressed. - `0` - otherwise.
Requires	Global variables: - `DriveA:` DriveA. - `DriveB:` DriveB. - `DriveA_Direction:` Direction of DriveA pin. - `DriveB_Direction:` Direction of DriveB pin. must be defined before using this function.

Example	`// Touch Panel module connections`
	`var DriveA : sbit at LATC13_bit;`
	`    DriveB : sbit at LATC14_bit;`
	`    DriveA_Direction : sbit at TRISC13_bit;`
	`    DriveB_Direction : sbit at TRISC14_bit;`
	`// End Touch Panel module connections`
	
	`if (TP_Press_Detect() <> 0) then`
	`  begin`
	`  ...`
	`  end;`

Notes	None.

TP_Get_Coordinates

Prototype	`function TP_Get_Coordinates(x_coordinate : ^word; y_coordinate : ^word) : byte;`
Description	Get touch panel coordinates and store them in `x_coordinate` and `y_coordinate` parameters.
Parameters	- `x_coordinate:` x coordinate of the place of touch. - `y_coordinate:` y coordinate of the place of touch.
Returns	- `1` - if reading is within display dimension range. - `0` - if reading is out of display dimension range.
Requires	Nothing.

Example	`if (TP_Get_Coordinates(@x_coord, @y_coord) = 0) then`
	`  begin`
	`  ...`
	`  end;`

Notes	None.

TP_Calibrate_Bottom_Left

Prototype	`procedure TP_Calibrate_Bottom_Left();`
Description	Calibrate bottom left corner of the touch Panel.
Parameters	None.
Returns	Nothing.
Requires	Nothing.
Example	`TP_Calibrate_Bottom_Left();    // Calibration of bottom left corner`
Notes	None.

TP_Calibrate_Upper_Right

Prototype	`procedure TP_Calibrate_Upper_Right();`
Description	Calibrate upper right corner of the touch panel.
Parameters	None.
Returns	Nothing.
Requires	Nothing.
Example	`TP_Calibrate_Upper_Right();    // Calibration of upper right corner`
Notes	None.

TP_Get_Calibration_Consts

Prototype	`procedure TP_Get_Calibration_Consts(x_min : ^word; x_max : ^word; y_min : ^word; y_max : ^word);`
Description	Gets calibration constants after calibration is done and stores them in `x_min, x_max, y_min` and `y_max` parameters.
Parameters	- `x_min`: x coordinate of the bottom left corner of the working area. - `x_max`: x coordinate of the upper right corner of the working area. - `y_min`: y coordinate of the bottom left corner of the working area. - `y_max`: y coordinate of the upper right corner of the working area.
Returns	Nothing.
Requires	Nothing.
Example	`TP_Get_Calibration_Consts(@x_min, @y_min, @x_max, @y_max);  // Get calibration constants`
Notes	None.

TP_Set_Calibration_Consts

Prototype	`procedure TP_Set_Calibration_Consts(x_min : word; x_max : word; y_min : word; y_max : word);`
Description	Sets calibration constants.
Parameters	- `x_min`: x coordinate of the bottom left corner of the working area. - `x_max`: x coordinate of the upper right corner of the working area. - `y_min`: y coordinate of the bottom left corner of the working area. - `y_max`: y coordinate of the upper right corner of the working area.
Returns	Nothing.
Requires	Nothing.
Example	`TP_Set_Calibration_Consts(148, 3590, 519, 3370);  // Set calibration constants`
Notes	None.

Touch Panel TFT Library

The mikroPascal PRO for PIC32 provides a library for working with Touch Panel for TFT.

Library Dependency Tree

Touch Panel → ADC

External dependencies of Touch Panel TFT Library

The following variables must be defined in all projects using Touch Panel TFT Library:	Description:	Example:
`var DriveX_Left : sbit; sfr; external;`	DriveX_Left line.	`var DriveX_Left : sbit at LATB13_bit;`
`var DriveX_Right : sbit; sfr; external;`	DriveX_Right line.	`var DriveX_Right : sbit at LATB11_bit;`
`var DriveY_Up : sbit; sfr; external;`	DriveY_Up line.	`var DriveY_Up : sbit at LATB12_bit;`
`var DriveY_Down : sbit; sfr; external;`	DriveY_Down line.	`var DriveY_Down : sbit at LATB10_bit;`
`var DriveX_Left_Direction : sbit; sfr; external;`	Direction of the DriveX_Left pin.	`var DriveX_Left_Direction : sbit at TRISB13_bit;`
`var DriveX_Right_Direction : sbit; sfr; external;`	Direction of the DriveX_Right pin.	`var DriveX_Right_Direction : sbit at TRISB11_bit;`
`var DriveY_Up_Direction : sbit; sfr; external;`	Direction of the DriveY_Up pin.	`var DriveY_Up_Direction : sbit at TRISB12_bit;`
`var DriveY_Down_Direction : sbit; sfr; external;`	Direction of the DriveY_Down pin.	`var DriveY_Down_Direction : sbit at TRISB10_bit;`

Library Routines

- TP_TFT_Init
- TP_TFT_Set_ADC_Threshold
- TP_TFT_Press_Detect
- TP_TFT_Get_Coordinates
- TP_TFT_Calibrate_Min
- TP_TFT_Calibrate_Max
- TP_TFT_Get_Calibration_Consts
- TP_TFT_Set_Calibration_Consts

TP_TFT_Init

Prototype	`procedure TP_TFT_Init(display_width : word; display_height : word; readX_ChNo : byte; readY_ChNo : byte);`
Description	Initialize TFT touch panel display. Default touch panel ADC threshold value is set to 900.
Parameters	- `display_width:` set display width. - `display_height:` set display height. - `readX_ChNo:` read X coordinate from desired ADC channel. - `readY_ChNo:` read Y coordinate from desired ADC channel.
Returns	Nothing.
Requires	Before calling this function initialize ADC module.

Example	`ADC1_Init();                    // Initalize ADC module`
	`TP_TFT_Init(320, 240, 13, 12); // Initialize touch panel`

Notes	None.

TP_TFT_Set_ADC_Threshold

Prototype	`procedure TP_TFT_Set_ADC_Threshold(threshold : word);`
Description	Set custom ADC threshold value, call this function after TP_TFT_Init.
Parameters	- `threshold`: custom ADC threshold value.
Returns	Nothing.
Requires	TP_TFT_Init has to be called before using this routine.
Example	`TP_TFT_Set_ADC_Threshold(900);    // Set touch panel ADC threshold`
Notes	None.

TP_TFT_Press_Detect

Prototype	`function TP_TFT_Press_Detect() : byte;`
Description	Detects if the touch panel has been pressed.
Parameters	None.
Returns	- `1` - if touch panel is pressed. - `0` - otherwise.
Requires	Global variables: - `DriveX_Left`: DriveX_Left pin. - `DriveX_Right`: DriveX_Right pin. - `DriveY_Up`: DriveY_Up pin. - `DriveY_Down`: DriveY_Down pin. - `DriveX_Left_Direction`: Direction of DriveX_Left pin. - `DriveX_Right_Direction`: Direction of DriveX_Right pin. - `DriveY_Up_Direction`: Direction of DriveY_Up pin. - `DriveY_Down_Direction`: Direction of DriveY_Down pin. must be defined before using this function.

Example	`// Touch Panel module connections`
	`var DriveX_Left : sbit at LATB13_bit;`
	`var DriveX_Right : sbit at LATB11_bit;`
	`var DriveY_Up : sbit at LATB12_bit;`
	`var DriveY_Down : sbit at LATB10_bit;`
	`var DriveX_Left_Direction : sbit at TRISB13_bit;`
	`var DriveX_Right_Direction : sbit at TRISB11_bit;`
	`var DriveY_Up_Direction : sbit at TRISB12_bit;`
	`var DriveY_Down_Direction : sbit at TRISB10_bit;`
	`// End Touch Panel module connections`
	
	`if (TP_TFT_Press_Detect() <> 0) then`
	`  begin`
	`  ...`
	`  end;`

Notes	None.

TP_TFT_Get_Coordinates

Prototype	`function TP_TFT_Get_Coordinates(x_coordinate : ^word; y_coordinate : ^word) : byte;`
Description	Get touch panel coordinates and store them in `x_coordinate` and `y_coordinate` parameters.
Parameters	- `x_coordinate`: x coordinate of the place of touch. - `y_coordinate`: y coordinate of the place of touch.
Returns	- `1` - if reading is within display dimension range. - `0` - if reading is out of display dimension range.
Requires	Nothing.

Example	`if (TP_TFT_Get_Coordinates(@x_coord, @y_coord) = 0) then`
	`  begin`
	`  ...`
	`  end;`

Notes	None.

TP_TFT_Calibrate_Min

Prototype	`procedure TP_TFT_Calibrate_Min();`
Description	Calibrate bottom left corner of the touch Panel.
Parameters	None.
Returns	Nothing.
Requires	Nothing.
Example	`TP_TFT_Calibrate_Min();    // Calibration of bottom left corner`
Notes	None.

TP_TFT_Calibrate_Max

Prototype	`procedure TP_TFT_Calibrate_Max();`
Description	Calibrate upper right corner of the touch panel.
Parameters	None.
Returns	Nothing.
Requires	Nothing.
Example	`TP_TFT_Calibrate_Max();    // Calibration of upper right corner`
Notes	None.

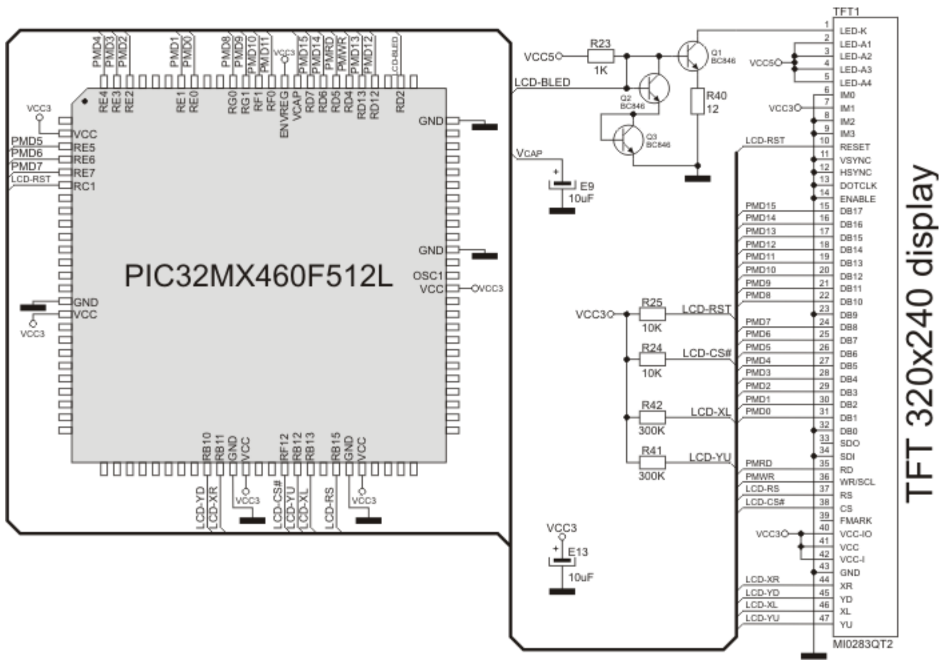
TP_TFT_Get_Calibration_Consts

Prototype	`procedure TP_TFT_Get_Calibration_Consts(x_min : ^word; x_max : ^word; y_min : ^word; y_max : ^word);`
Description	Gets calibration constants after calibration is done and stores them in `x_min`, `x_max`, `y_min` and `y_max` parameters.
Parameters	- `x_min`: x coordinate of the bottom left corner of the working area. - `x_max`: x coordinate of the upper right corner of the working area. - `y_min`: y coordinate of the bottom left corner of the working area. - `y_max`: y coordinate of the upper right corner of the working area.
Returns	Nothing.
Requires	Nothing.
Example	`TP_TFT_Get_Calibration_Consts(@x_min, @y_min, @x_max, @y_max);     // Get calibration constants`
Notes	None.

TP_TFT_Set_Calibration_Consts

Prototype	`procedure TP_TFT_Set_Calibration_Consts(x_min : word; x_max : word; y_min : word; y_max : word);`
Description	Sets calibration constants.
Parameters	- `x_min`: x coordinate of the bottom left corner of the working area. - `x_max`: x coordinate of the upper right corner of the working area. - `y_min`: y coordinate of the bottom left corner of the working area. - `y_max`: y coordinate of the upper right corner of the working area.
Returns	Nothing.
Requires	Nothing.
Example	`TP_TFT_Set_Calibration_Consts(173, 776, 75, 760);     // Set calibration constants`
Notes	None.

HW Connection



Touch Panel TFT HW connection

UART Library

The UART hardware module is available with a number of PIC32 MCUs. The mikroPascal PRO for PIC32 UART Library provides comfortable work with the Asynchronous (full duplex) mode.

Important:

- UART library routines require you to specify the module you want to use. To select the desired UART module, simply change the letter **x** in the routine prototype for a number from **1** to **6**.
- Number of UART modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.
- Switching between the UART modules in the UART library is done by the UART_Set_Active function (UART modules have to be previously initialized).

Library Routines

- UARTx_Init
- UARTx_Init_Advanced
- UARTx_Data_Ready
- UARTx_Tx_Idle
- UARTx_Read
- UARTx_Read_Text
- UARTx_Write
- UARTx_Write_Text
- UART_Set_Active

UARTx_Init

Prototype	`procedure UARTx_Init(baud_rate : dword);`
Description	Configures and initializes the UART module. The internal UART module module is set to: - continue operation in IDLE mode - default Tx and Rx pins - loopback mode disabled - 8-bit data, no parity - 1 STOP bit - transmitter enabled - generate interrupt on transmission end - interrupt on reception enabled - Address Detect mode disabled
Parameters	- `baud_rate`: requested baud rate
Returns	Nothing.
Requires	Routine requires the UART module.
Example	`// Initialize hardware UART1 module and establish communication at 2400 bps` `UART1_Init(2400);`
Notes	- Refer to the device data sheet for baud rates allowed for specific Fosc. - The compiler will choose for which speed the calculation is to be performed (high or low). This does not mean that it is the best choice for desired baud rate. If the baud rate error generated in this way is too big then UARTx_Init_Advanced routine, which allows speed select be used. - UART library routines require you to specify the module you want to use. To select the desired UART module, simply change the letter **x** in the routine prototype for a number from **1** to **6**. - Switching between the UART modules in the UART library is done by the UART_Set_Active function (UART modules have to be previously initialized). - Number of UART modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

UARTx_Init_Advanced

Prototype	`procedure UARTx_Init_Advanced(baud_rate, freq_Khz : dword; high_low, parity, stop_bits : byte);`
Description	Configures and initializes the UART module with user defined settings.
Parameters	- `baud_rate`: requested baud rate - `freq_Khz`: Peripheral Bus Clock frequency in kHz. - `high_low_speed`: High/low speed selection parameter. Valid values: (see High/Low Speed table) - `parity`: Parity and data selection parameter. Valid values: (see Data/Parity Mode table) - `stop_bits`: stop bit selection parameter. Valid values: (see Stop bits table)
Returns	Nothing.
Requires	Routine requires the UART module.
Example	
Notes	- Refer to the device data sheet for baud rates allowed for specific Fosc. - UART library routines require you to specify the module you want to use. To select the desired UART module, simply change the letter **x** in the routine prototype for a number from **1** to **6**. - Switching between the UART modules in the UART library is done by the UART_Set_Active function (UART modules have to be previously initialized). - Number of UART modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

High/Low Speed

Description	Predefined library const
`Low Speed UART`	`_UART_LOW_SPEED`
`Hi Speed UART`	`_UART_HI_SPEED`

Data/Parity Mode

Description	Predefined library const
`8-bit data, no parity`	`_UART_8BIT_NOPARITY`
`8-bit data, even parity`	`_UART_8BIT_EVENPARITY`
`8-bit data, odd parity`	`_UART_8BIT_ODDPARITY`
`9-bit data, no parity`	`_UART_9BIT_NOPARITY`

Stop bits

Description	Predefined library const
`One stop bit`	`_UART_ONE_STOPBIT`
`Two stop bit`	`_UART_TWO_STOPBITS`

UARTx_Data_Ready

Prototype	`function UARTx_Data_Ready() : word;`
Description	The function tests if data in receive buffer is ready for reading.
Parameters	None.
Returns	- `1` if data is ready for reading - `0` if there is no data in the receive register
Requires	Routine requires at least one UART module. Used UART module must be initialized before using this routine. See UARTx_Init and UARTx_Init_ Advanced routines.

Example	`var receive : word;`
	`...`
	`// read data if ready`
	`if (UART1_Data_Ready() = 1) then`
	`  receive := UART1_Read();`

Notes	- UART library routines require you to specify the module you want to use. To select the desired UART module, simply change the letter **x** in the routine prototype for a number from **1** to **6**. - Number of UART modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

UARTx_Tx_Idle

Prototype	`function UARTx_Tx_Idle() : word;`
Description	Use the function to test if the transmit shift register is empty or not.
Parameters	None.
Returns	- `1` if the data has been transmitted - `0` otherwise
Requires	Routine requires at least one UART module. Used UART module must be initialized before using this routine. See UARTx_Init and UARTx_Init_ Advanced routines.

Example	`// If the previous data has been shifted out, send next data:`
	`if (UART1_Tx_Idle() = 1) then`
	`  UART1_Write(_data);`

Notes	- UART library routines require you to specify the module you want to use. To select the desired UART module, simply change the letter **x** in the routine prototype for a number from **1** to **6**. - Number of UART modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

UARTx_Read

Prototype	`function UARTx_Read() : word;`
Description	The function receives a byte via UART. Use the UARTx_Data_Ready function to test if data is ready first.
Parameters	None.
Returns	Received byte.
Requires	Routine requires at least one UART module. Used UART module must be initialized before using this routine. See UARTx_Init and UARTx_Init_Advanced routines.

Example	`var receive : word;`
	`...`
	`// read data if ready`
	`if (UART1_Data_Ready() = 1) then`
	`  receive := UART1_Read();`

Notes	- UART library routines require you to specify the module you want to use. To select the desired UART module, simply change the letter **x** in the routine prototype for a number from **1** to **6**. - Number of UART modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

UARTx_Read_Text

Prototype	`procedure UARTx_Read_Text(var output, delimiter : string; Attempts : byte);`
Description	Reads characters received via UART until the delimiter sequence is detected. The read sequence is stored in the parameter `output`; delimiter sequence is stored in the parameter `delimiter`. This is a blocking call: the delimiter sequence is expected, otherwise the procedure exits (if the delimiter is not found).
Parameters	- `Output`: received text - `Delimiter`: sequence of characters that identifies the end of a received string - `Attempts`: defines number of received characters in which `Delimiter` sequence is expected. If `Attempts` is set to 255, this routine will continuously try to detect the `Delimiter` sequence.
Returns	Nothing.
Requires	Routine requires at least one UART module. Used UART module must be initialized before using this routine. See UARTx_Init and UARTx_Init_Advanced routines.

Example	Read text until the sequence “OK” is received, and send back what’s been received:
	
	`// Read text until the sequence “OK” is received, and then send it back:`
	`UART1_Init(9600);`
	`delim := ‘OK’;`
	`while TRUE do`
	`begin`
	`  if UART1_Data_Ready() = 1 then`
	`  begin`
	`    UART1_Read_Text(txt, delim, 10);`
	`    UART1_Write_Text(txt);`
	`  end;`
	`end;`

Notes	- UART library routines require you to specify the module you want to use. To select the desired UART module, simply change the letter **x** in the routine prototype for a number from **1** to **6**. - Number of UART modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

UARTx_Write

Prototype	`procedure UARTx_Write(data_ : word);`
Description	The function transmits a byte via the UART module.
Parameters	- `data`: data to be sent
Returns	Nothing.
Requires	Routine requires at least one UART module. Used UART module must be initialized before using this routine. See UARTx_Init and UARTx_Init_Advanced routines.

Example	`var data_ : byte;`
	`...`
	`data_ := 0x1E;`
	`UART1_Write(data_);`

Notes	- UART library routines require you to specify the module you want to use. To select the desired UART module, simply change the letter **x** in the routine prototype for a number from **1** to **6**. - Number of UART modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

UARTx_Write_Text

Prototype	`procedure UARTx_Write_Text(var uart_text : string);`
Description	Sends text via UART. Text should be zero terminated.
Parameters	- `UART_text`: text to be sent
Returns	Nothing.
Requires	Routine requires at least one UART module. Used UART module must be initialized before using this routine. See UARTx_Init and UARTx_Init_Advanced routines.

Example	Read text until the sequence “OK” is received, and send back what’s been received:
	
	`// Read text until the sequence “OK” is received, and then send it back:`
	`UART1_Init(9600);`
	`delim := ‘OK’;`
	`while TRUE do`
	`begin`
	`  if UART1_Data_Ready() = 1 then`
	`  begin`
	`    UART1_Read_Text(txt, delim, 10);`
	`    UART1_Write_Text(txt);`
	`  end;`
	`end;`

Notes	- UART library routines require you to specify the module you want to use. To select the desired UART module, simply change the letter **x** in the routine prototype for a number from **1** to **6**. - Number of UART modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

UART_Set_Active

Prototype	`procedure UART_Set_Active (read_ptr : ^TUART_Rd_Ptr; write_ptr : ^TUART_Wr_Ptr; ready_ptr : ^TUART_Rdy_Ptr; tx_idle_ptr : ^TUART_TX_Idle_Ptr);`
Description	Sets active UART module which will be used by UARTx_Data_Ready, UARTx_Read and UARTx_Write routines.
Parameters	Parameters: - `read_ptr`: UARTx_Read handler - `write_ptr`: UARTx_Write handler - `ready_ptr`: UARTx_Data_Ready handler - `tx_idle_ptr`: UARTx_Tx_Idle handler
Returns	Nothing.
Requires	Routine is available only for MCUs with multiple UART modules. Used UART module must be initialized before using this routine. See UARTx_Init and UARTx_Init_Advanced routines.

Example	`UART1_Init(9600);                    // initialize UART1 module`
	`UART2_Init(9600);                    // initialize UART2 module`
	
	`RS485Master_Init();                  // initialize MCU as Master`
	
	`UART_Set_Active(@UART1_Read, @UART1_Write, @UART1_Data_Ready, @UART1_Tx_Idle); // set UART1 active`
	`RS485Master_Send(dat,1,160);        // send message through UART1`
	
	`UART_Set_Active(@UART2_Read, @UART2_Write, @UART2_Data_Ready, @UART2_Tx_Idle); // set UART2 active`
	`RS485Master_Send(dat,1,160);        // send through UART2`

Notes	None.

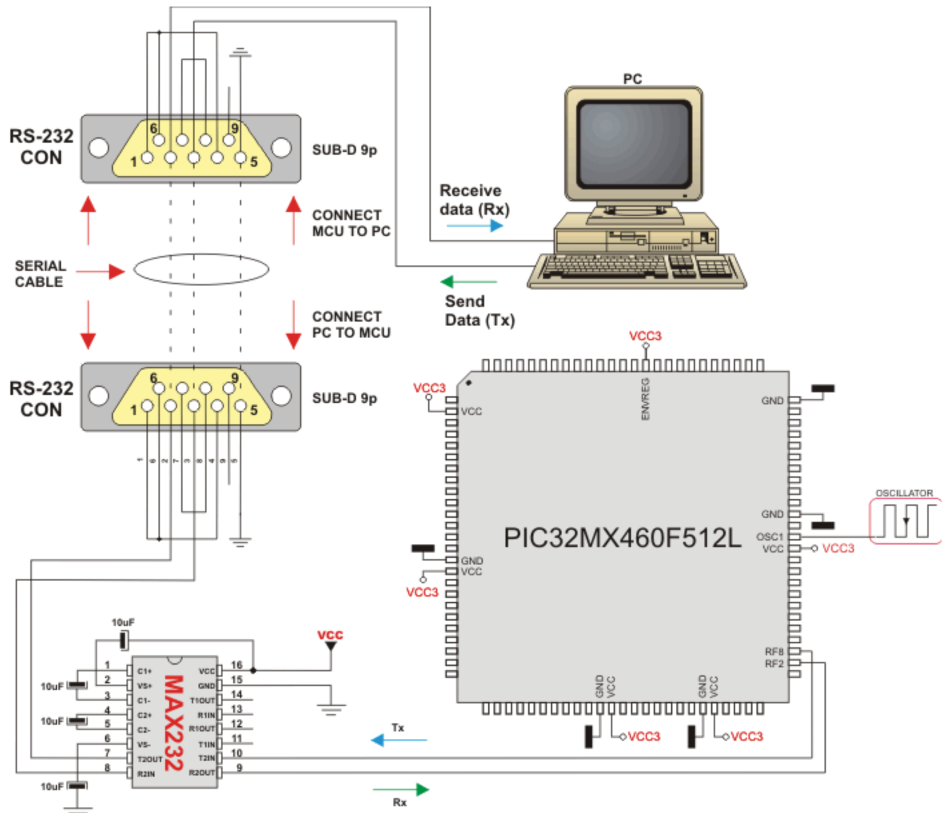
Library Example

This example demonstrates simple data exchange via UART. If MCU is connected to the PC, you can test the example from the mikroPascal PRO for PIC32 USART communication terminal, launch it from the drop-down menu **Tools › USART Terminal** or simply click the USART Terminal Icon .

Copy Code To Clipboard

```
program UART1;
var uart_rd : byte;

begin

  CHECON := 0x32;
  AD1PCFG := 0xFFFF;                       // Configure AN pins as digital I/O

  UART1_Init(56000);                       // Initialize UART module at 56000 bps
  Delay_ms(100);                           // Wait for UART module to stabilize

  UART1_Write_Text('Start');
  UART1_Write(13);
  UART1_Write(10);

  while (TRUE) do                          // Endless loop
    begin
      if (UART1_Data_Ready() <> 0) then    // If data is received
        begin
          uart_rd := UART1_Read();         // read the received data
          UART1_Write(uart_rd);            // and send data via UART
        end;
    end;
end.
```

HW Connection



RS232 HW connection

USB Library

Universal Serial Bus (USB) provides a serial bus standard for connecting a wide variety of devices, including computers, cell phones, game consoles, PDA's, etc.

USB Library contains HID routines that support HID class devices, and also the generic routines that can be used with vendor specified drivers.

USB HID Class

The HID class consists primarily of devices that are used by humans to control the operation of computer systems. Typical examples of HID class devices include:

- Keyboards and pointing devices, for example: standard mouse devices, trackballs, and joysticks.
- Front-panel controls, for example: knobs, switches, buttons, and sliders.
- Controls that might be found on devices such as telephones, VCR remote controls, games or simulation devices, for example: data gloves, throttles, steering wheels, and rudder pedals.
- Devices that may not require human interaction but provide data in a similar format to HID class devices, for example, bar-code readers, thermometers, or voltmeters.

Many typical HID class devices include indicators, specialized displays, audio feedback, and force or tactile feedback. Therefore, the HID class definition includes support for various types of output directed to the end user.

Descriptor File

Each project based on the USB library should include a descriptor source file which contains vendor id and name, product id and name, report length, and other relevant information. To create a descriptor file, use the integrated USB HID terminal of mikroPascal PRO for PIC32(Tools › USB HID Terminal). The default name for descriptor file is USBdsc.mpas, but you may rename it.

Library Routines

- HID_Enable
- HID_Read
- HID_Write
- HID_Disable
- USB_Interrupt_Proc
- USB_Polling_Proc
- Gen_Enable
- Gen_Read
- Gen_Write

HID_Enable

Prototype	`procedure HID_Enable(readbuff : ^byte; writebuff : ^byte);`
Description	Enables USB HID communication.
Parameters	- `readbuff`: Read Buffer. - `writebuff`: Write Buffer. These parameters are used for HID communication.
Returns	Nothing.
Requires	Nothing
Example	`HID_Enable(&readbuff,&writebuff);`
Notes	This function needs to be called before using other routines of USB HID Library.

HID_Read

Prototype	`function HID_Read() : byte;`
Description	Receives message from host and stores it in the Read Buffer.
Parameters	None.
Returns	If the data reading has failed, the function returns 0. Otherwise, it returns number of characters received from the host.
Requires	USB HID needs to be enabled before using this function. See HID_Enable.

Example	`// retry until success`
	`while(HID_Read() = 0) do`
	`  ;`

Notes	None.

HID_Write

Prototype	`function HID_Write(writebuff : ^byte; len : byte) : byte;`
Description	Function sends data from Write Buffer writebuff to host.
Parameters	- `writebuff`: Write Buffer, same parameter as used in initialization; see HID_Enable. - `len`: specifies a length of the data to be transmitted.
Returns	If the data transmitting has failed, the function returns 0. Otherwise, it returns number of transmitted bytes.
Requires	USB HID needs to be enabled before using this function. See HID_Enable.

Example	`// retry until success`
	`while(HID_Write(@writebuff,64) = 0) do`
	`  ;`

Notes	Function call needs to be repeated as long as data is not successfuly sent.

HID_Disable

Prototype	`procedure HID_Disable();`
Description	Disables USB HID communication.
Parameters	None.
Returns	Nothing.
Requires	USB HID needs to be enabled before using this function. See HID_Enable.
Example	`HID_Disable();`
Notes	None.

USB_Interrupt_Proc

Prototype	`procedure USB_Interrupt_Proc();`
Description	This routine is used for servicing various USB bus events. Should be called inside USB interrupt routine.
Parameters	None.
Returns	Nothing.
Requires	Nothing.

Example	`procedure USB1Interrupt(); iv IVT_ADDR_USB1INTERRUPT;`
	`begin`
	`  USB_Interrupt_Proc();`
	`end;`

Notes	Do not use this function with USB_Polling_Proc, only one should be used. To enable servicing through interrupt, `USB_INTERRUPT` constant should be set (it is set by default in descriptor file).

USB_Polling_Proc

Prototype	`procedure USB_Polling_Proc();`
Description	This routine is used for servicing various USB bus events. It should be periodically, preferably every 100 microseconds.
Parameters	None.
Returns	Nothing.
Requires	Nothing.

Example	`while TRUE do`
	`begin`
	`  USB_Polling_Proc();`
	`  kk := HID_Read();`
	`  if (kk <> 0) then`
	`  begin`
	`    for cnt := 0 to 64`
	`      writebuff[cnt] := readbuff[cnt];`
	`    HID_Write(@writebuff,64);`
	`  end;`
	`end;`

Notes	Do not use this functions with USB_Interrupt_Proc. To enable servicing by polling, `USB_INTERRUPT` constant should be set to 0 (it is located in descriptor file).

Gen_Enable

Prototype	`procedure Gen_Enable(readbuff : ^byte; writebuff : ^byte);`
Description	Initialize the USB module of the MCU.
Parameters	- `readbuff`: Read Buffer. - `writebuff`: Write Buffer.
Returns	Nothing.
Requires	USB needs to be enabled before using this function. See HID_Enable.
Example	`Gen_Enable(&readbuff,&writebuff);`
Notes	None.

Gen_Read

Prototype	`function Gen_Read(readbuff : ^byte; length : byte; ep : byte) : byte;`
Description	Generic routine that receives the specified data from the specified endpoint.
Parameters	- `readbuff`: Received data. - `length`: The length of the data that you wish to receive. - `ep`: Endpoint number you want to receive the data into.
Returns	Returns the number of received bytes, otherwise 0.
Requires	USB needs to be enabled before using this function. See HID_Enable.
Example	`while(Gen_Read(@readbuff,64,1) = 0) do` `  ;`
Notes	None.

Gen_Write

Prototype	`function Gen_Write(writebuff : ^byte; length : byte; ep : byte) : byte;`
Description	Sends the specified data to the specified endpoint.
Parameters	- `writebuff`: The data that you want to send. - `length`: the length of the data that you wish to send. - `ep`: Endpoint number you want to send the data into.
Returns	Returns the number of transmitted bytes, otherwise 0.
Requires	USB needs to be enabled before using this function. See HID_Enable.
Example	`while(Gen_Write(@writebuff,64,1) = 0) do` `  ;`
Notes	None.

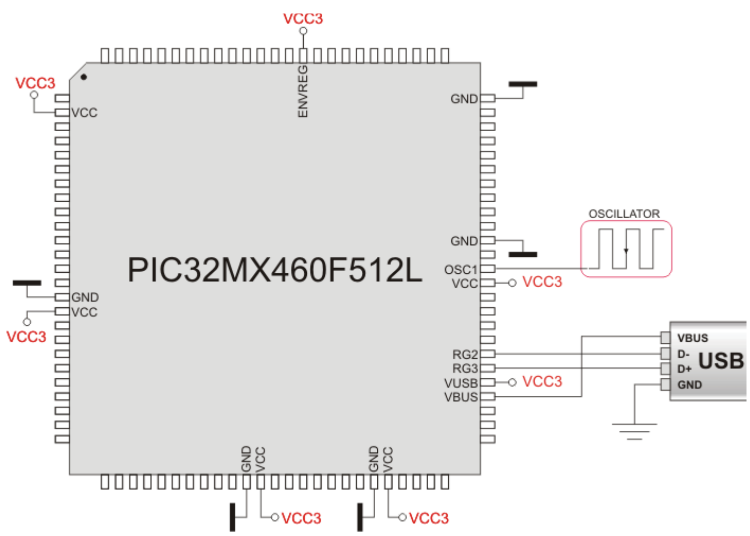
Library Example

This example establishes connection with the HID terminal that is active on the PC. Upon connection establishment, the HID Device Name will appear in the respective window. After that software will wait for data and it will return received data back. Examples uses USBdsc.mpas descriptor file, which is in the same folder, and can be created by the HID Terminal.

Copy Code To Clipboard

```
program HID_Read_Write_Polling;

var cnt, kk : char;

var readbuff : array[64] of char;
var writebuff : array[64] of char;

begin
  CHECON := 0x32;
  AD1PCFG := 0xFFFF;
  HID_Enable(@readbuff,@writebuff);
  while TRUE do
    begin
      USB_Polling_Proc();                // Call this routine periodically
      kk := HID_Read();
      if (kk <> 0) then
      begin
        for cnt:=0 to 63 do
          writebuff[cnt]:=readbuff[cnt];
        HID_Write(@writebuff,64);
      end ;
    end;
end.
```

HW Connection



USB connection scheme

Miscellaneous Libraries

- Button Library
- C Type Library
- Conversions Library
- Setjmp Library
- String Library
- Time Library
- Trigon Library
- Trigonometry Library

Button Library

The Button Library provides routines for detecting button presses and debouncing (eliminating the influence of contact flickering upon pressing a button).

Library Routines

- Button

Button

Prototype	`function Button(var port: word; pin: byte; time: word; ActiveState: byte) : word;`
Description	The function eliminates the influence of contact flickering upon pressing a button (debouncing). The Button pin is tested just after the function call and then again after the debouncing period has expired. If the pin was in the active state in both cases then the function returns 255 (true).
Parameters	- `port:` button port address - `pin:` button pin - `time:` debouncing period in milliseconds - `active_state:` determines what is considered as active state. Valid values: `0` (logical zero) and `1` (logical one)
Returns	- `255` if the pin was in the active state for given period. - `0` otherwise
Requires	Nothing.

Example	`program Button_Test;`
	
	`var oldstate : bit;`
	
	`begin`
	`  oldstate := 0;`
	`  ADPCFG := 0xFFFF;                    // initialize AN pins as digital`
	`  TRISD  := 0xFFFF;                    // initialize PORTD as input`
	`  TRISB  := 0x0000;                    // initialize PORTB as output`

Button

Example	<pre> while TRUE do
	    begin
	      if (Button(PORTD, 0, 1, 1)) then // detect logical
	one on RB0 pin
	        oldstate := 1;
	      if (oldstate and Button(PORTD, 0, 1, 0)) then
	        begin // detect one-
	to-zero transition on RB0 pin
	          LATB := not LATB;
	          oldstate := 0;
	        end;
	    end; // endless loop
	end.</pre>

Notes	None.

C Type Library

The mikroPascal PRO for PIC32 provides a set of library functions for testing and mapping characters.

Library Functions

- isalnum
- isalpha
- iscntrl
- isdigit
- isgraph
- islower
- ispunct
- isspace
- isupper
- isxdigit
- toupper
- tolower

isalnum

Prototype	`function isalnum(character : byte) : word`
Description	Function returns 0xFF if the character is alphanumeric (A-Z, a-z, 0-9), otherwise returns zero.

Example	`res := isalnum('o');    // returns 0xFF`
	`res := isalnum('\r');   // returns 0`

isalpha

Prototype	`function isalpha(character : byte) : word`
Description	Function returns 0xFF if the character is alphabetic (A-Z, a-z), otherwise returns zero.

Example	`res := isalpha('A');  // returns 0xFF`
	`res := isalpha('1');  // returns 0`

iscntrl

Prototype	`function iscntrl(character : byte) : word`
Description	Function returns 0xFF if the character is a control or delete character(decimal 0-31 and 127), otherwise returns zero.

Example	`res := iscntrl('\r');  // returns 0xFF`
	`res := iscntrl('o');   // returns 0`

isdigit

Prototype	`function isdigit(character : byte) : word`
Description	Function returns 0xFF if the character is a digit (0-9), otherwise returns zero.

Example	`res := isdigit('o');  // returns 0xFF`
	`res := isdigit('1');  // returns 0`

isgraph

Prototype	`function isgraph(character : byte) : word`
Description	Function returns 0xFF if the `character` is a printable, excluding the space (decimal 32), otherwise returns zero.

Example	`res := isgraph('o');  // returns 0xFF`
	`res := isgraph(' ');  // returns 0`

islower

Prototype	`function islower(character : byte) : word`
Description	Function returns 0xFF if the character is a lowercase letter (a-z), otherwise returns zero.

Example	`res := islower('0');  // returns 0xFF`
	`res := islower('A');  // returns 0`

ispunct

Prototype	`function ispunct(character : byte) : word`
Description	Function returns 0xFF if the `character` is a punctuation (decimal 32-47, 58-63, 91-96, 123-126), otherwise returns zero.

Example	`res := islower('0');  // returns 0xFF`
	`res := islower('A');  // returns 0`

isspace

Prototype	`function isspace(character : byte) : word`
Description	Function returns 0xFF if the `character` is a white space (space, tab, CR, HT, VT, NL, FF), otherwise returns zero.

Example	`res := isspace(' ');  // returns 0xFF`
	`res := isspace('1');  // returns 0`

isupper

Prototype	`function isupper(character : byte) : word`
Description	Function returns 0xFF if the `character` is an uppercase letter (A-Z), otherwise returns zero.

Example	`res := isupper('A');  // returns 0xFF`
	`res := isupper('a');  // returns 0`

isxdigit

Prototype	`function isxdigit(character : byte) : word`
Description	Function returns 0xFF if the `character` is a hex digit (0-9, A-F, a-f), otherwise returns zero.

Example	`res := isxdigit('A');  // returns 0xFF`
	`res := isxdigit('P');  // returns 0`

toupper

Prototype	`function toupper(character : byte) : byte`
Description	If the `character` is a lowercase letter (a-z), the function returns an uppercase letter. Otherwise, the function returns an unchanged input parameter.

Example	`res := toupper('a');  // returns A`
	`res := toupper('B');  // returns B`

tolower

Prototype	`function tolower(character : byte) : byte`
Description	If the `character` is an uppercase letter (A-Z), function returns a lowercase letter. Otherwise, function returns an unchanged input parameter.

Example	`res := tolower('A');  // returns a`
	`res := tolower('b');  // returns b`

Conversions Library

mikroPascal PRO for PIC32 Conversions Library provides routines for numerals to strings and BCD/decimal conversions.

Library Dependency Tree

Library Routines

You can get text representation of numerical value by passing it to one of the following routines:

- ByteToStr
- ShortToStr
- WordToStr
- IntToStr
- LongToStr
- LongWordToStr
- FloatToStr

- WordToStrWithZeros
- IntToStrWithZeros
- LongWordToStrWithZeros
- LongIntToStrWithZeros

- ByteToHex
- ShortToHex
- WordToHex
- IntToHex
- LongWordToHex
- LongIntToHex

- StrToInt
- StrToWord

The following functions convert decimal values to BCD and vice versa:

- Bcd2Dec
- Dec2Bcd
- Bcd2Dec16
- Dec2Bcd16

ByteToStr

Prototype	`procedure ByteToStr(input : byte; var output : array[3] of char);`
Description	Converts input byte to a string. The output string is right justified and remaining positions on the left (if any) are filled with blanks.
Parameters	- `input:` byte to be converted - `output:` destination string
Returns	Nothing.
Requires	Nothing.

Example	`var t : byte;`
	`    txt : array[3] of char;`
	`...`
	`t := 24;`
	`ByteToStr(t, txt);  // txt is " 24" (one blank here)`

Notes	None.

ShortToStr

Prototype	`procedure ShortToStr(input : short; var output : array[4] of char);`
Description	Converts input short (signed byte) number to a string. The output string is right justified and remaining positions on the left (if any) are filled with blanks.
Parameters	- `input:` signed short number to be converted - `output:` destination string
Returns	Nothing.
Requires	Destination string should be at least 5 characters in length.

Example	`var t : short;`
	`    txt : array[4] of char;`
	`...`
	`t := -24;`
	`ByteToStr(t, txt);  // txt is " -24" (one blank here)`

Notes	None.

WordToStr

Prototype	`procedure WordToStr(input : word; var output : array[5] of char);`
Description	Converts input word to a string. The output string is right justified and the remaining positions on the left (if any) are filled with blanks.
Parameters	- `input`: word to be converted - `output`: destination string
Returns	Nothing.
Requires	Nothing.

Example	`var t : word;`
	`    txt : array[5] of char;`
	`...`
	`t := 437;`
	`WordToStr(t, txt);  // txt is "  437" (two blanks here)`

Notes	None.

IntToStr

Prototype	`procedure IntToStr(input : integer; var output : array[6] of char);`
Description	Converts input integer number to a string. The output string is right justified and the remaining positions on the left (if any) are filled with blanks.
Parameters	- `input:` signed integer number to be converted - `output:` destination string
Returns	Nothing.
Requires	Nothing.

Example	`var input : integer;`
	`    txt : array[6] of char;`
	`//...`
	`begin`
	`input := -4220;`
	`IntToStr(input, txt);   // txt is ' -4220'`

Notes	None.

LongToStr

Prototype	`procedure LongintToStr(input : longint; var output : array[11] of char);`
Description	Converts input longint number to a string. The output string is right justified and the remaining positions on the left (if any) are filled with blanks.
Parameters	- `input`: signed long integer number to be converted - `output`: destination string
Returns	Nothing.
Requires	Nothing.

Example	`var input : longint;`
	`    txt : array[11] of char;`
	`//...`
	`begin`
	`input := -12345678;`
	`IntToStr(input, txt);    // txt is '  -12345678'`

Notes	None.

LongWordToStr

Prototype	`procedure LongWordToStr(input : dword; var output : array[10] of char);`
Description	Converts input double word number to a string. The output string is right justified and the remaining positions on the left (if any) are filled with blanks.
Parameters	- `input`: unsigned long integer number to be converted - `output`: destination string
Returns	Nothing.
Requires	Nothing.

Example	`var input : longint;`
	`    txt : array[10] of char;`
	`//...`
	`begin`
	`input := 12345678;`
	`IntToStr(input, txt);    // txt is '  12345678'`

Notes	None.

FloatToStr

Prototype	`procedure FloatToStr(fnum : real; var str : array[23] of char) : byte;`
Description	Converts a floating point number to a string. The output string is left justified and null terminated after the last digit.
Parameters	- `fnum`: floating point number to be converted - `str`: destination string
Returns	Nothing.
Requires	Nothing.

Example	<pre>var ff1, ff2, ff3 : real;
	    txt : array[10] of char;
	  ...
	  ff1 := -374.2;
	  ff2 := 123.456789;
	  ff3 := 0.000001234;
	
	  FloatToStr(ff1, txt); // txt is "-374.20001"
	  FloatToStr(ff2, txt); // txt is "123.45678"
	  FloatToStr(ff3, txt); // txt is "0.000000"</pre>

Notes	Given floating point number will be truncated to 7 most significant digits before conversion.

WordToStrWithZeros

Prototype	`procedure WordToStrWithZeros(input: word; var output: array[5] of char);`
Description	Converts input word to a string. The output string is right justified and the remaining positions on the left (if any) are filled with zeros.
Parameters	- `input`: unsigned integer to be converted - `output`: destination string
Returns	Nothing.
Requires	Nothing.

Example	<pre>var t : word;
	    txt : array[5] of char;
	//...
	t := 437;
	WordToStrWithZeros(t, txt); // txt is '00437'</pre>

Notes	None.

IntToStrWithZeros

Prototype	`procedure IntToStrWithZeros(input: integer; var output: array[6] of char);`
Description	Converts input integer to a string. The output string is right justified and the remaining positions on the left (if any) are filled with zeros.
Parameters	- `input`: integer number to be converted - `output`: destination string
Returns	Nothing.
Requires	Nothing.

Example	`var t : integer;`
	`    txt : array[6] of char;`
	`//...`
	`t := -3276;`
	`IntToStrWithZeros(t, txt);  // txt is '-03276'`

Notes	None.

LongWordToStrWithZeros

Prototype	`procedure LongWordToStrWithZeros(input: dword; var output: array[10] of char);`
Description	Converts input dword to a string. The output string is right justified and the remaining positions on the left (if any) are filled with zeros.
Parameters	- `input`: unsigned long number to be converted - `output`: destination string
Returns	Nothing.
Requires	Nothing.

Example	`var t : dword;`
	`    txt : array[10] of char;`
	`//...`
	`t := 12345678;`
	`LongWordToStrWithZeros(t, txt);  // txt is '0012345678'`

Notes	None.

LongIntToStrWithZeros

Prototype	`procedure LongIntToStrWithZeros(input: longint; var output: array[11] of char);`
Description	Converts input longint to a string. The output string is right justified and the remaining positions on the left (if any) are filled with zeros.
Parameters	- `input`: signed long number to be converted - `output`: destination string
Returns	Nothing.
Requires	Nothing.

Example	`var t : longint;`
	`    txt : array[11] of char;`
	`//...`
	`t := -12345678;`
	`LongIntToStrWithZeros(t, txt);  // txt is '-0012345678'`

Notes	None.

ByteToHex

Prototype	`procedure ByteToHex(input : byte; var output : array[2] of char);`
Description	Converts input number to a string containing the number's hexadecimal representation. The output string is right justified and remaining positions on the left (if any) are filled with zeros.
Parameters	- `input`: byte to be converted - `output`: destination string
Returns	Nothing.
Requires	Nothing.

Example	`var t : byte;`
	`    txt : array[2] of char;`
	`...`
	`t := 2;`
	`ByteToHex(t, txt);  // txt is "02"`

Notes	None.

ShortToHex

Prototype	`procedure ShortToHex(input : short; var output : array[2] of char);`
Description	Converts input number to a string containing the number's hexadecimal representation. The output string is right justified and remaining positions on the left (if any) are filled with zeros.
Parameters	- `input`: signed short number to be converted - `output`: destination string
Returns	Nothing.
Requires	Nothing.

Example	`var t : short;`
	`    txt : array[2] of char;`
	`...`
	`t := -100;`
	`ShortToHex(t, txt);  // txt is "9C"`

Notes	None.

WordToHex

Prototype	`procedure WordToHex(input : word; var output : array[4] of char);`
Description	Converts input number to a string containing the number's hexadecimal representation. The output string is right justified and remaining positions on the left (if any) are filled with zeros.
Parameters	- `input`: unsigned integer to be converted - `output`: destination string
Returns	Nothing.
Requires	Nothing.

Example	`var t : word;`
	`    txt : array[4] of char;`
	`...`
	`t := 1111;`
	`WordToHex(t, txt);  // txt is "0457"`

Notes	None.

IntToHex

Prototype	`procedure IntToHex(input : integer; var output : array[64] of char);`
Description	Converts input number to a string containing the number's hexadecimal representation. The output string is right justified and remaining positions on the left (if any) are filled with zeros.
Parameters	- `input`: signed integer number to be converted - `output`: destination string
Returns	Nothing.
Requires	Nothing.

Example	`var input : integer;`
	`    txt : string[4];`
	`//...`
	`input := -32768;`
	`IntToHex(input, txt);   // txt is '8000'`

Notes	None.

LongWordToHex

Prototype	`procedure LongWordToHex(input : dword; var output : array[8] of char);`
Description	Converts input number to a string containing the number's hexadecimal representation. The output string is right justified and remaining positions on the left (if any) are filled with zeros.
Parameters	- `input`: unsigned long integer number to be converted - `output`: destination string
Returns	Nothing.
Requires	Nothing.

Example	`var input : dword;`
	`    txt : array[8] of char;`
	`//...`
	`input := 65535;`
	`LongWordToHex(input, txt);    // txt is '0000FFFF'`

Notes	None.

LongIntToHex

Prototype	`procedure LongIntToHex(input : longint; var output : array[8] of char);`
Description	Converts input number to a string containing the number's hexadecimal representation. The output string is right justified and remaining positions on the left (if any) are filled with zeros.
Parameters	- `input`: signed long integer number to be converted - `output`: destination string
Returns	Nothing.
Requires	Nothing.

Example	`var input : longint;`
	`    txt : array[8] of char;`
	`//...`
	`input := -2147483648;`
	`LongIntToHex(input, txt);    // txt is '80000000'`

Notes	None.

StrToInt

Prototype	`function StrToInt(var input: string[6]): integer;`
Description	Converts a string to an integer.
Parameters	- `input`: string to be converted
Returns	Integer variable.
Requires	Input string is assumed to be the correct representation of a number. The conversion will end with the first character which is not a decimal digit.

Example	`var ii: integer;`
	
	`begin`
	`  ii:= StrToInt('-1234');`
	`end.`

Notes	None.

StrToWord

Prototype	`function StrToWord(var input: string[5]): word;`
Description	Converts a string to word.
Parameters	- `input`: string to be converted
Returns	Word variable.
Requires	Input string is assumed to be the correct representation of a number. The conversion will end with the first character which is not a decimal digit.

Example	`var ww: word;`
	
	`begin`
	`  ww:= StrToword('65432');`
	`end.`

Notes	None.

Bcd2Dec

Prototype	`function Bcd2Dec(bcdnum : byte) : byte;`
Description	Converts input BCD number to its appropriate decimal representation.
Parameters	- `bcdnum`: number to be converted
Returns	Converted decimal value.
Requires	Nothing.

Example	`var a, b : byte;`
	`...`
	`a := 22;`
	`b := Bcd2Dec(a); // b equals 34`

Notes	None.

Dec2Bcd

Prototype	`function Dec2Bcd(decnum : byte) : byte;`
Description	Converts input unsigned short integer number to its appropriate BCD representation.
Parameters	- `decnum`: number to be converted
Returns	Converted BCD value.
Requires	Nothing.

Example	`var a, b : byte;`
	`...`
	`a := 22;`
	`b := Dec2Bcd(a); // b equals 34`

Notes	None.

Bcd2Dec16

Prototype	`function Bcd2Dec16(bcdnum : word) : word;`
Description	Converts unsigned 16-bit decimal value to its BCD equivalent.
Parameters	- `bcdnum`: 16-bit BCD numeral to be converted
Returns	Converted decimal value.
Requires	Nothing.

Example	`var a, b : word;`
	`...`
	`a := 0x1234;         // a equals 4660`
	`b := Bcd2Dec16(a);   // b equals 1234`

Notes	None.

Dec2Bcd16

Prototype	`function Dec2Bcd16(decnum : word) : word;`
Description	Converts decimal value to its BCD equivalent.
Parameters	- `decnum` decimal number to be converted
Returns	Converted decimal value.
Requires	Nothing.

Example	`var a, b : word;`
	`...`
	`a := 2345;`
	`b := Dec2Bcd16(a);   // b equals 9029`

Notes	None.

Setjmp Library

The Setjmp library contains functions and types definitions for bypassing the normal function call and return discipline.

Library Routines

- Setjmp
- Longjmp

Setjmp

Prototype	`function setjmp(var env : array[4] of word) : integer;`
Returns	- `0` if the return is from direct invocation - `nonzero value` if the return is from a call to `longjmp` (this value will be set by the `longjmp` routine)
Description	This function saves calling position for a later use by longjmp. Parameters : - `env`: buffer suitable for holding information needed for restoring calling environment
Requires	Nothing.

Example	`var buf : array[4] of word;`
	`...`
	`Setjmp(buf);`

Longjmp

Prototype	`procedure longjmp(var env : array[4] of word; val : integer);`
Returns	Nothing.
Description	Restores calling environment saved in the env buffer by the most recent invocation of setjmp. If there has been no such invocation, or the function containing the invocation of setjmp has terminated in the interim, the behavior is undefined. Parameters : - `env`: buffer holding the information saved by the corresponding `setjmp` invocation - `val`: value to be returned by the corresponding setjmp function
Requires	Invocation of `longjmp` must occur before return from the function in which `setjmp` was called encounters.

Example	`var buf : array[4] of word;`
	`...`
	`Longjmp(buf, 2);`

Library Example

Example demonstrates function cross calling using setjmp and longjmp functions. When called, Setjmp() saves its calling environment in its **buf** argument for later use by the Longjmp(). Longjmp(), on the other hand, restores the environment saved by the most recent invocation of the Setjmp() with the corresponding **buf** argument.

Copy Code To Clipboard

```
program Setjmp;

var buf : array[4] of word ;   //  Note : Program flow diagrams are indexed according
                               //  to the sequence of execution

procedure func33();            //  2<------------|
  begin                        //                |
    Delay_ms(1000);            //                |
                               //                |
    nop;                       //                |
    longjmp(buf, 2);           //  3---------------->|
    nop;                       //                |   |
                               //                |   |
  end;                         //                |   |
                               //                |   |
procedure func();              //  1<--------|   |   |
  begin                        //            |   |   |
    PORTB := 3;                //            |   |   |
    if (setjmp(buf) = 2) then  //  3<----------------|
      PORTB := 1               //  4-->|     |   |
    else                       //      |     |   |
      func33();                //  2------------>|
                               //      |     |
                               //  4<--|     |
  end;                         //  5----->|  |
                               //         |  |
begin                          //         |  |
    ADPCFG := 0xFFFF;          //         |  |
                               //         |  |
    PORTB := 0;                //         |  |
    TRISB := 0;                //         |  |
                               //         |  |
    nop;                       //         |  |
                               //         |  |
    func();                    //  1-------->|
                               //         |
    nop;                       //  5<-----|
    Delay_ms(1000);
    PORTB := 0xFFFF;
end.
```

Sprint Library

mikroPascal PRO for PIC32 includes a library which automatizes string related tasks.

Library Dependency Tree

Functions

- memchr
- memcmp
- memcpy
- memmove
- memset
- strcat
- strcat2
- strchr
- strcmp
- strcpy
- strlen
- strncat
- strncat2
- strncpy
- strspn
- strncmp
- strstr
- strcspn
- strpbrk
- strrchr
- ltrim
- rtrim
- strappendpre
- strappendsuf
- length

memchr

Prototype	`function memchr(p : ^byte; ch : byte; n : word) : word;`
Description	The function locates the first occurrence of the word ch in the initial n words of memory area starting at the address p. The function returns the offset of this occurrence from the memory address p or `0xFFFF` if ch was not found. For the parameter p you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example `@mystring` or `@PORTB`.

Example	`txt := 'mikroElektronika';`
	
	`res := memchr(@txt, 'e', 16);  // example locates first occurrence of the letter 'e' in the string 'txt' in the first 16 characters of the string`

memcmp

Prototype	`function memcmp(p1, p2 : ^byte; n : word) : integer;`

Description	The function returns a positive, negative, or zero value indicating the relationship of first n words of memory areas starting at addresses `p1` and `p2`.
	
	This function compares two memory areas starting at addresses p1 and p2 for n words and returns a value indicating their relationship as follows:
	
	Value **Meaning**
	`< 0` `p1 "less than" p2`
	`= 0` `p1 "equal to" p2`
	`> 0` `p1 "greater than" p2`
	The value returned by the function is determined by the difference between the values of the first pair of words that differ in the strings being compared.
	
	For parameters p1 and p2 you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example `@mystring` or `@PORTB`.

Example	`txt := 'mikroElektronika';`
	`txt_sub := 'mikro';`
	
	`res := memcmp(@txt, @txt_sub, 16);  // returns 69, which is ASCII code of the first differing character - letter 'E'`

memcpy

Prototype	`procedure memcpy(p1, p2 : ^byte; nn : word);`
Description	The function copies nn words from the memory area starting at the address `p2` to the memory area starting at p1. If these memory buffers overlap, the `memcpy` function cannot guarantee that words are copied before being overwritten. If these buffers do overlap, use the `memmove` function. For parameters `p1` and `p2` you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example `@mystring` or `@PORTB`.

Example	`txt := 'mikroElektronika';`
	`txt_sub := 'mikr';`
	
	`memcpy(@txt+4, @txt_sub, 4); // string 'txt' will be populated with the first 4 characters of the 'txt_sub' string, beginning from the 4th character`

memmove

Prototype	`procedure memmove(p1, p2 : ^byte; nn : word);`
Description	The function copies nn words from the memory area starting at the address `p2` to the memory area starting at `p1`. If these memory buffers overlap, the `Memmove` function ensures that the words in `p2` are copied to `p1` before being overwritten. For parameters `p1` and `p2` you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example `@mystring` or `@PORTB`.

Example	`txt := 'mikroElektronika';`
	`txt_sub := 'mikr';`
	
	`memmove(@txt+7, @txt_sub, 4); // string 'txt' will be populated with first 4 characters of the 'txt_sub' string, beginning from the 7th character`

memset

Prototype	`procedure memset(p : ^byte; character : byte; n : word);`
Description	The function fills the first n words in the memory area starting at the address p with the value of word `character`. For parameter `p` you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example `@mystring` or `@PORTB`.

Example	`txt := 'mikroElektronika';`
	
	`memset(@txt, 'a', 2); // routine will copy the character 'a' into each of the first 'n' characters of the string 'txt',`

strcat

Prototype	`procedure strcat(var s1, s2 : string);`
Description	The function appends the value of string `s2` to string `s1` and terminates `s1` with a null character.

Example	`txt := 'mikroElektronika';`
	`txt_sub := 'mikr';`
	
	`txt[3] := 0;`
	`strcat(txt, '_test');  // routine will append the '_test' at the place of the first null character, adding terminating null character to the result`

strcat2

Prototype	`procedure strcat2(var l1, s1, s2 : string);`
Description	The procedure adjoins string `s2` at the end of the string `s1`, or at the first null character of the `s1`, and places the result string into l string.

Example	`txt := 'mikroElektronika';`
	`txt_sub := '_Test';`
	`l1 := string[21];`
	
	`strcat2(l1, txt, txt_sub);  // routine will adjoin strings txt and txt_sub and place the result into l; l = mikroElektronika_Test`

strchr

Prototype	`function strchr(var s : string; ch : byte) : word;`
Description	The function searches the string `s` for the first occurrence of the character `ch`. The null character terminating s is not included in the search. The function returns the position (index) of the first character `ch` found in `s`; if no matching character was found, the function returns `0xFFFF`.

Example	`txt := 'mikroElektronika';`
	
	`res := strchr(txt, 'E');  // routine will locate the character 'E' in the 'txt' string, and return the position of the character`

strcmp

Prototype	`function strcmp(var s1, s2 : string) : integer;`

Description	The function lexicographically compares the contents of the strings `s1` and `s2` and returns a value indicating their relationship:
	
	Value **Meaning**
	`< 0` `s1 "less than" s2`
	`= 0` `s1 "equal to" s2`
	`> 0` `s1 "greater than" s2`
	The value returned by the function is determined by the difference between the values of the first pair of words that differ in the strings being compared.

Example	`txt := 'mikroElektronika';`
	
	`res := strchr(txt, 'E');  // routine will locate the character 'E' in the 'txt' string, and return the position of the character`

strcpy

Prototype	`procedure strcpy(var s1, s2 : string);`
Description	The function copies the value of the string `s2` to the string `s1` and appends a null character to the end of `s1`.

Example	`txt := 'mikroElektronika';`
	`txt_sub := 'mikr';`
	
	`strcpy(txt,txt_sub);  // copies string 'txt_sub' to 'txt'`

strlen

Prototype	`function strlen(var s : string) : word;`
Description	The function returns the length, in words, of the string `s`. The length does not include the null terminating character.

Example	`txt := 'mikroElektronika';`
	
	`res = strlen(txt);  // calculates the length of the 'txt' string, result = 16`

strncat

Prototype	`procedure strncat(var s1, s2 : string; size : word);`
Description	The function appends at most size characters from the string `s2` to the string `s1` and terminates `s1` with a null character. If `s2` is shorter than the size characters, `s2` is copied up to and including the null terminating character.

Example	`txt := 'mikroElektronika';`
	`txt_sub := 'mikr';`
	`txt[5] := 0;`
	
	`strncat(txt,txt_sub,4);  // routine appends first 4 characters from the string 'txt_sub' at the place of first null character in the 'txt' string`

strncpy

Prototype	`procedure strncpy(var s1, s2 : string; size : word);`
Description	The function copies at most size characters from the string `s2` to the string `s1`. If `s2` contains fewer characters than `size`, `s1` is padded out with null characters up to the total length of the `size` characters.

Example	`txt := 'mikroElektronika';`
	`txt_sub := 'mikr';`
	
	`strncpy(txt,txt_sub,4);  // copies first 4 characters form the string 'txt_sub' to 'txt'`

strspn

Prototype	`function strspn(var s1, s2 : string) : word;`
Description	The function searches the string `s1` for characters not found in the `s2` string. The function returns the index of first character located in `s1` that does not match a character in `s2`. If the first character in `s1` does not match a character in `s2`, a value of 0 is returned. If all characters in `s1` are found in `s2`, the length of `s1` is returned (not including the terminating null character).

Example	`txt := 'mikroElektronika';`
	`txt_sub := 'mikr';`
	
	`res := strspn(txt,txt_sub);  // routne returns 4`

strncmp

Prototype	`function strncmp(var s1, s2 : string; len : word) : integer;`

Description	The function lexicographically compares the first len characters of the strings `s1` and `s2` and returns a value indicating their relationship:
	
	`Value      Meaning`
	`< 0        s1 "less than" s2`
	`= 0        s1 "equal to" s2`
	`> 0        s1 "greater than" s2`
	The value returned by the function is determined by the difference between the values of the first pair of words that differ in the strings being compared (within first `len` words).

Example	`txt := 'mikroElektronika';`
	`txt_sub := 'mikr';`
	
	`res := strncmp(txt_sub,txt,3);  // compares the first 3 characters from the string 'txt' with the sting 'txt_sub' and returns a difference`

strstr

Prototype	`function strstr(var s1, s2 : string) : word;`
Description	The function locates the first occurrence of the string `s2` in the string `s1` (excluding the terminating null character). The function returns a number indicating the position of the first occurrence of `s2` in `s1`; if no string was found, the function returns 0xFFFF. If s2 is a null string, the function returns `0`.

Example	`txt := 'mikroElektronika';`
	`txt_sub := 'mikr';`
	
	`res := strstr(txt_sub,txt);`

strcspn

Prototype	`function strcspn(var s1, s2 : string) : word;`
Description	The function searches the string `s1` for any of the characters in the string `s2`. The function returns the index of the first character located in `s1` that matches any character in `s2`. If the first character in `s1` matches a character in `s2`, a value of 0 is returned. If there are no matching characters in `s1`, the length of the string is returned (not including the terminating null character).

Example	`txt := 'mikroElektronika';`
	`txt_sub := 'mikr';`
	
	`res := strcspn(txt_sub,txt);`

strpbrk

Prototype	`function strpbrk(var s1, s2 : string) : word;`
Description	The function searches `s1` for the first occurrence of any character from the string `s2`. The null terminator is not included in the search. The function returns an index of the matching character in `s1`. If `s1` contains no characters from `s2`, the function returns `0xFFFF`.

Example	`txt := 'mikroElektronika';`
	`txt_sub := 'mikr';`
	
	`res := strpbrk(txt_sub,txt);`

strrchr

Prototype	`function strrchr(var s : string; ch : byte) : word;`
Description	The function searches the string s for the last occurrence of the character `ch`. The null character terminating s is not included in the search. The function returns an index of the last ch found in `s`; if no matching character was found, the function returns `0xFFFF`.

Example	`txt := 'mikroElektronika';`
	
	`res = strrchr(txt,'k'); // returns the index of the 'k' character of the 'txt' string`

ltrim

Prototype	`procedure ltrim(var astring : string);`
Description	The procedure trims the leading spaces of the string.

Example	`txt := '  mikroE';`
	
	`ltrim(txt);  // trims the leading 2 spaces of the 'txt' string`

rtrim

Prototype	`procedure rtrim(var astring : string);`
Description	The procedure trims the trailing spaces of the string.

Example	`txt := 'mikroE  ';`
	
	`rtrim(txt);  // trims the trailing 2 spaces of the 'txt' string and adds terminating null character to the result`

strappendpre

Prototype	`procedure strappendpre(letter: char; var s1 : string);`
Description	The procedure appends character at the beginning of the string.

Example	`txt := 'ikroE';`
	
	`strappendpre('m',txt);  // adds letter 'm' at the beginning of the 'txt' string`

strappendsuf

Prototype	`procedure strappendsuf(var s1 : string; letter : char);`
Description	The procedure appends character at the end of the string.

Example	`txt := 'mikro';`
	
	`strappendsuf('E',txt);  // adds letter 'E' at the end of the 'txt' string`

length

Prototype	`function length(var s: string) : word;`
Description	The function returns length of passed string.

Example	`txt := 'mikroE';`
	
	`res = length(txt);  // calculates and returns the length of the 'txt' string`

Time Library

The Time Library contains functions and type definitions for time calculations in the UNIX time format which counts the number of seconds since the "epoch". This is very convenient for programs that work with time intervals: the difference between two UNIX time values is a real-time difference measured in seconds.

What is the epoch?
Originally it was defined as the beginning of 1970 GMT. (January 1, 1970 Julian day) GMT, Greenwich Mean Time, is a traditional term for the time zone in England.

The TimeStruct type is a structure type suitable for time and date storage.

Library Routines

- Time_dateToEpoch
- Time_epochToDate
- Time_dateDiff

Time_dateToEpoch

Prototype	`function Time_dateToEpoch(var ts : TimeStruct) : longint;`
Description	This function returns the UNIX time : number of seconds since January 1, 1970 0h00mn00s.
Parameters	- `ts`: time and date value for calculating UNIX time.
Returns	Number of seconds since January 1, 1970 0h00mn00s.
Requires	Nothing.

Example	`var ts1 : TimeStruct;`
	`    Epoch : longint;`
	`...`
	`// what is the epoch of the date in ts ?`
	`epoch := Time_dateToEpoch(@ts1) ;`

Notes	None.

Time_epochToDate

Prototype	`procedure Time_epochToDate(e : longint; var ts : TimeStruct);`
Description	Converts the UNIX time to time and date.
Parameters	- `e`: UNIX time (seconds since UNIX epoch) - `ts`: time and date structure for storing conversion output
Returns	Nothing.
Requires	Nothing.

Example	`var ts2 : TimeStruct;`
	`    epoch : longint;`
	`...`
	`//what date is epoch 1234567890 ?`
	`epoch := 1234567890 ;`
	`Time_epochToDate(epoch,ts2);`

Notes	None.

Time_dateDiff

Prototype	`function Time_dateDiff(var t1, t2 : TimeStruct) : longint ;`
Description	This function compares two dates and returns time difference in seconds as a signed long. Result is positive if `t1` is before `t2`, result is null if `t1` is the same as `t2` and result is negative if `t1` is after `t2`.
Parameters	- `t1`: time and date structure (the first comparison parameter) - `t2`: time and date structure (the second comparison parameter)
Parameters	None.
Returns	Time difference in seconds as a signed long.
Requires	Nothing.

Example	`var ts1, ts2 : TimeStruct;`
	`    diff : longint;`
	`...`
	`//how many seconds between these two dates contained in ts1 and ts2 buffers?`
	` diff := Time_dateDiff(ts1, ts2);`

Notes	None.

Library Example

Demonstration of Time library routines usage for time calculations in UNIX time format.

Copy Code To Clipboard

```
program Time_Demo;
  {*
   * simple time structure
   *}
  type TimeStruct = record
      ss : byte ;     // seconds
      mn : byte ;     // minutes
```

```
     hh : byte ;    // hours
     md : byte ;    // day in month, from 1 to 31
     wd : byte ;    // day in week, monday=0, tuesday=1, .... sunday=6
     mo : byte ;    // month number, from 1 to 12 (and not from 0 to 11 as with unix C
time !)
     yy : word ;    // year Y2K compliant, from 1892 to 2038
  end;

  var  ts1, ts2    : TimeStruct;
       buf         : array[256] of byte ;
       epoch, diff : longint ;

  begin
    ts1.ss := 0 ;
    ts1.mn := 7 ;
    ts1.hh := 17 ;
    ts1.md := 23 ;
    ts1.mo := 5 ;
    ts1.yy := 2006 ;

    {*
     * what is the epoch of the date in ts ?
     *}
    epoch := Time_dateToEpoch(@ts1) ;   // epoch = 1148404020

    {*
     * what date is epoch 1234567890 ?
     *}

    epoch := 1234567890 ;
    Time_epochToDate(epoch, @ts2) ;    // ts2.ss := 30 ;
                                       // ts2.mn := 31 ;
                                       // ts2.hh := 23 ;
                                       // ts2.md := 13 ;
                                       // ts2.wd := 4 ;
                                       // ts2.mo := 2 ;
                                       // ts2.yy := 2009 ;

    {*
     * how much seconds between this two dates ?
     *}
    diff := Time_dateDiff(@ts1, @ts2) ;  // diff = 86163870

  end.
```

TimeStruct type definition

```
type TimeStruct = record
     ss : byte ;    // seconds
     mn : byte ;    // minutes
     hh : byte ;    // hours
     md : byte ;    // day in month, from 1 to 31
     wd : byte ;    // day in week, monday=0, tuesday=1, .... sunday=6
     mo : byte ;    // month number, from 1 to 12 (and not from 0 to 11 as with unix C
time !)
     yy : word ;    // year Y2K compliant, from 1892 to 2038
  end;
```

Trigon Library

The mikroPascal PRO for PIC32 provides a set of library functions for floating point math handling. See also Predefined Globals and Constants for the list of predefined math constants.

Library Functions

- acos
- asin
- atan
- atan2
- ceil
- cos
- cosh
- eval_poly
- exp
- fabs
- floor
- frexp
- ldexp
- log
- log10
- modf
- pow
- sin
- sinh
- sqrt
- tan
- tanh

acos

Prototype	`function acos(x : real) : real;`
Description	Function returns the arc cosine of parameter `x`; that is, the value whose cosine is `x`. The input parameter `x` must be between -1 and 1 (inclusive). The return value is in radians, between 0 and Π (inclusive).
Example	`res := acos(0.5);  // res := 1.047198`

asin

Prototype	`function asin(x : real) : real;`
Description	Function returns the arc sine of parameter `x`; that is, the value whose sine is `x`. The input parameter `x` must be between -1 and 1 (inclusive). The return value is in radians, between -Π/2 and Π/2 (inclusive).
Example	`res := asin(0.5);  // res := 5.235987e-1`

atan

Prototype	`function atan(arg : real) : real;`
Description	Function computes the arc tangent of parameter `f`; that is, the value whose tangent is `f`. The return value is in radians, between -Π/2 and Π/2 (inclusive).
Example	`res := atan(1.0);  // res := 7.853982e-1`

atan2

Prototype	`function atan2(y : real; x : real) : real;`
Description	This is the two-argument arc tangent function. It is similar to computing the arc tangent of `y/x`, except that the signs of both arguments are used to determine the quadrant of the result and `x` is permitted to be zero. The return value is in radians, between -Π and Π (inclusive).
Example	`res := atan2(2., 1.);  // res := 4.636475e-1`

ceil

Prototype	`function ceil(x : real) : real;`
Description	Function returns value of parameter x rounded up to the next whole number.
Example	`res := ceil(0.5);  // res := 1.000000`

cos

Prototype	`function cos(arg : real) : real;`
Description	Function returns the cosine of f in radians. The return value is from -1 to 1.
Example	`res := cos(PI/3.);  // res := 0.500008`

cosh

Prototype	`function cosh(x : real) : real;`
Description	Function returns the hyperbolic cosine of `x`, defined mathematically as $(e^{x}+e^{-x})/2$. If the value of `x` is too large (if overflow occurs), the function fails.
Example	`res := cosh(PI/3.);  // res := 1.600286`

eval_poly

Prototype	`function eval_poly(x : real; var d : array[10] of real; n : byte) : real;`
Description	Function Calculates polynom for number `x`, with coefficients stored in `d[]`, for degree `n`.

exp

Prototype	`function exp(x : real) : real;`
Description	Function returns the value of e — the base of natural logarithms — raised to the power `x` (i.e. e^x).
Example	`res := exp(0.5);   // res := 1.648721`

fabs

Prototype	`function fabs(d : real) : real;`
Description	Function returns the absolute (i.e. positive) value of `d`.
Example	`res := fabs(-1.3);  // res := 1.3`

floor

Prototype	`function floor(x : real) : real;`
Description	Function returns the value of parameter x rounded down to the nearest integer.
Example	`res := floor(15.258);  // res := 15.000000`

frexp

Prototype	`function frexp(value : real; var eptr : integer) : real;`
Description	The function splits a floating-point value `value` into a normalized fraction and an integral power of 2. The return value is a normalized fraction and the integer exponent is stored in the object pointed to by `eptr`.

ldexp

Prototype	`function ldexp(value : real; newexp : integer) : real;`
Description	Function returns the result of multiplying the floating-point number `num` by 2 raised to the power n (i.e. returns $x * 2^n$).
Example	`res := ldexp(2.5, 2);  // res := 10`

log

Prototype	`function log(x : real) : real;`
Description	Function returns the natural logarithm of x (i.e. $\log_e(x)$).
Example	`res := log(10);  // res := 2.302585E`

log10

Prototype	`function log10(x : real) : real;`
Description	Function returns the base-10 logarithm of x (i.e. $\log_{10}(x)$).
Example	`res := log10(100.);  // res := 2.000000`

modf

Prototype	`function modf(val : real; var iptr : real) : real;`
Description	Returns argument val split to the fractional part (function return val) and integer part (in number iptr).
Example	`res := modf(6.25, iptr);  // res := 0.25, iptr = 6.00`

pow

Prototype	`function pow(x : real; y : real) : real;`
Description	Function returns the value of x raised to the power y (i.e. x^y). If x is negative, the function will automatically cast y into `unsigned long`.
Example	`res := pow(10.,5.);  // res := 9.999984e+4`

sin

Prototype	`function sin(arg : real) : real;`
Description	Function returns the sine of `f` in radians. The return value is from -1 to 1.
Example	`res := sin(PI/2.);  // res := 1.000000`

sinh

Prototype	`function sinh(x : real) : real;`
Description	Function returns the hyperbolic sine of `x`, defined mathematically as $(e^x-e^{-x})/2$. If the value of `x` is too large (if overflow occurs), the function fails.
Example	`res := sinh(PI/2.);  // res := 2.301296`

sqrt

Prototype	`function sqrt(x : real) : real;`
Description	Function returns the non negative square root of `x`.
Example	`res := sqrt(10000.);   // res := 100.0000`

tan

Prototype	`function tan(x : real) : real;`
Description	Function returns the tangent of x in radians. The return value spans the allowed range of floating point in the mikroPascal PRO for PIC32.
Example	`res := tan(PI/4.);  // res := 0.999998`

tanh

Prototype	`function tanh(x : real) : real;`
Description	Function returns the hyperbolic tangent of `x`, defined mathematically as `sinh(x)/cosh(x)`.
Example	`res := tanh(-PI/4.);   // res := -0.655793`

Trigonometry Library

The mikroPascal PRO for PIC32 implements fundamental trigonometry functions. These functions are implemented as look-up tables. Trigonometry functions are implemented in integer format in order to save memory.

Library Routines

- sinE3
- cosE3

sinE3

Prototype	`function sinE3(angle_deg : word): integer;`
Description	The function calculates sine multiplied by 1000 and rounded to the nearest integer: `result = round(sin(angle_deg)*1000)`
Parameters	- `angle_deg`: input angle in degrees
Returns	The function returns the sine of input parameter multiplied by 1000.
Requires	Nothing.

Example	`var res : integer;`
	`...`
	`res := sinE3(45);  // result is 707`

Notes	Return value range: `-1000..1000.`

cosE3

Prototype	`function cosE3(angle_deg : word): integer;`
Description	The function calculates cosine multiplied by 1000 and rounded to the nearest integer: `result = round(cos(angle_deg)*1000)`
Parameters	- `angle_deg`: input angle in degrees
Returns	The function returns the cosine of input parameter multiplied by 1000.
Requires	Nothing.

Example	`var res: integer;`
	`...`
	`res := cosE3(196);  // result is -193`

Notes	Return value range: `-1000..1000.`

CHAPTER 10

Tutorials

Managing Project

Projects

The mikroPascal PRO for PIC32 organizes applications into projects, consisting of a single project file (extension `.mpp32`) and one or more source files (extension `.mpas`). mikroPascal PRO for PIC32 IDE allows you to manage multiple projects (see Project Manager). Source files can be compiled only if they are part of a project.

The project file contains the following information:

- project name and optional description,
- target device,
- device flags (config word),
- device clock,
- list of the project source files with paths,
- binary files (*.emcl),
- image files,
- other files.

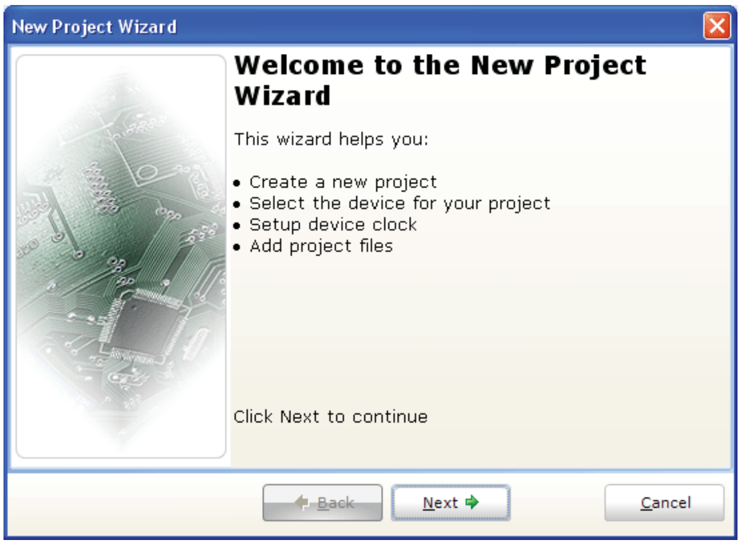
Note that the project does not include files in the same way as preprocessor does, see Add/Remove Files from Project.

New Project

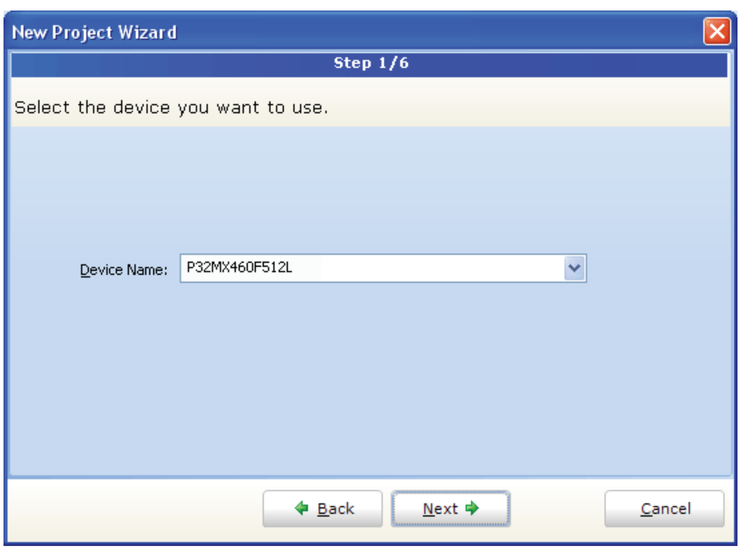
The easiest way to create a project is by means of the New Project Wizard, drop-down menu **Project › New Project** or by clicking the New Project Icon from Project Toolbar.

New Project Wizard Steps

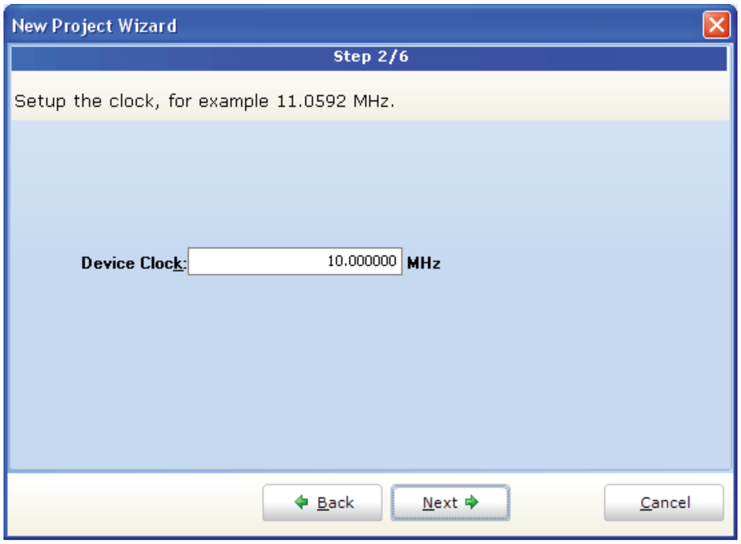
Start creating your New project, by clicking Next button:



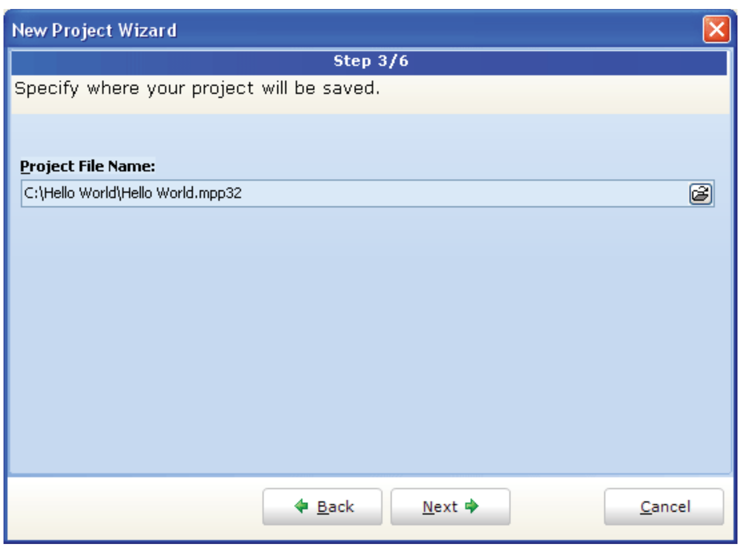
Step One - Select the device from the device drop-down list:



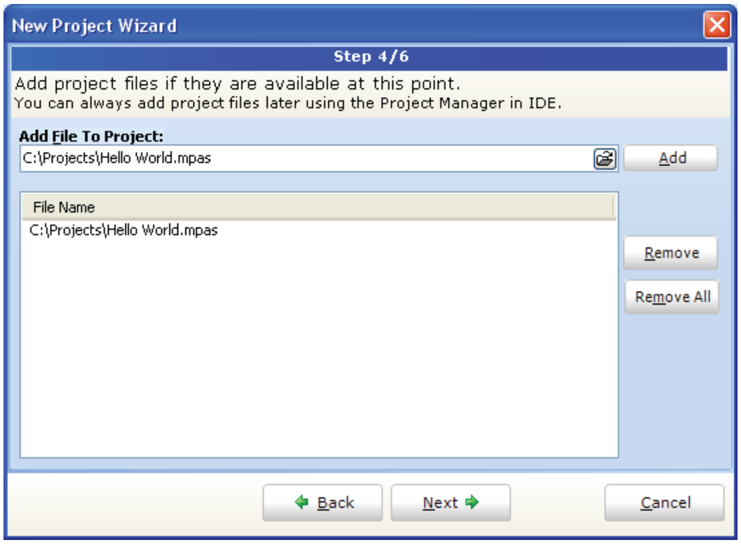
Step Two - Enter the oscillator frequency value:



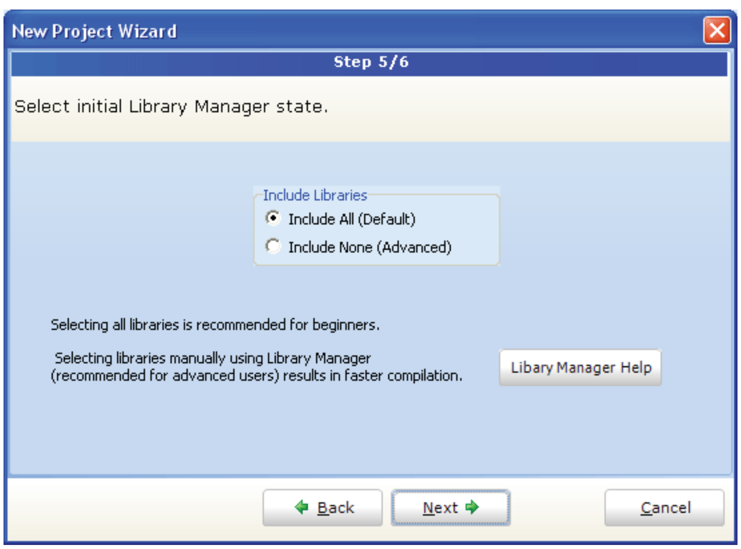
Step Three - Specify the location where your project will be saved:



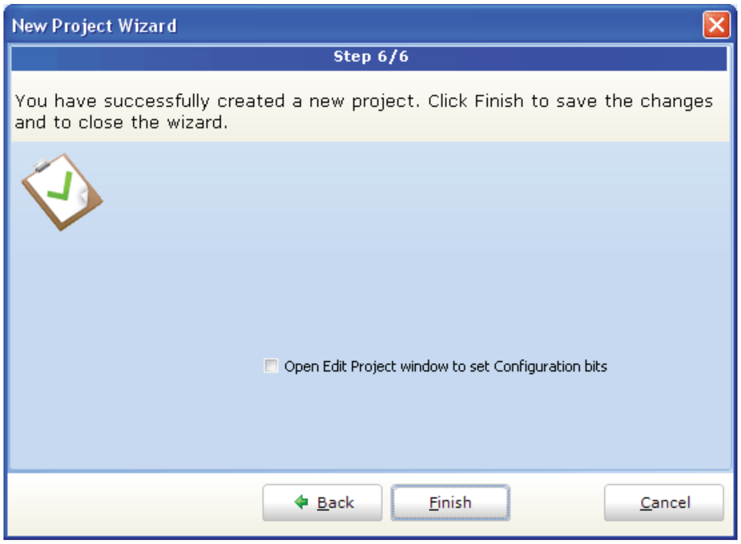
Step Four - Add project file to the project if they are avaiable at this point. You can always add project files later using Project Manager:



Step Five - Select inital Library Manager state:



Step Six - Click Finish button to create your New Project:



Related topics: Project Manager, Project Settings

Customizing Projects

You can change basic project settings in the Project Settings window. You can change chip and oscillator frequency. Any change in the Project Setting Window affects currently active project only, so in case more than one project is open, you have to ensure that exactly the desired project is set as active one in the Project Manager.

Also, you can change configuration bits of the selected chip in the Edit Project window.

Managing Project Group

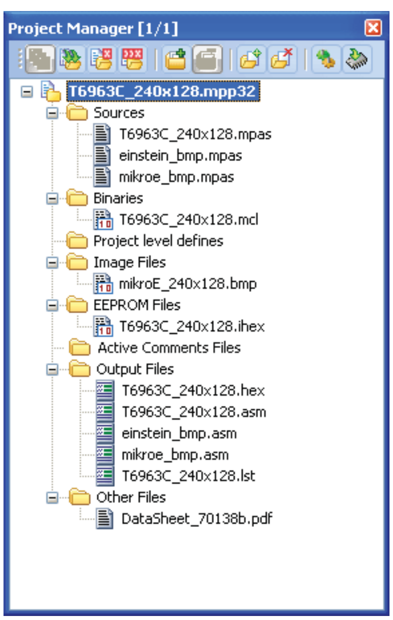
mikroPascal PRO for PIC32 IDE provides convenient option which enables several projects to be open simultaneously. If you have several projects being connected in some way, you can create a project group.

The project group may be saved by clicking the Save Project Group Icon from the Project Manager window. The project group may be reopened by clicking the Open Project Group Icon . All relevant data about the project group is stored in the project group file (extension `.mpdsgroup`)

Add/Remove Files from Project

The project can contain the following file types:

- `.mpas` source files
- `.emcl` binary files
- `.pld` project level defines files
- image files
- `.ihex` EEPROM files
- `.hex`, `.asm` and `.lst` files, see output files. These files can not be added or removed from project.
- other files



The list of relevant files is stored in the project file (extension `.mpp32`).

To add a file to the project, click the Add File to Project Icon or press Insert button on your keyboard. Each added source file must be self-contained, i.e. it must have all necessary definitions after preprocessing.

To remove file(s) from the project, click the Remove File from Project Icon or press Delete button on your keyboard.

Project Level Defines:

Project Level Defines (`.pld`) files can also be added to project. Project level define files enable you to have defines that are visible in all source files in the project. A file must contain one definition per line in the following form:

```
ANALOG
DEBUG
TEST
```

For example, lets make a project level define named `pld_test`. First of all, create a new file with the `.pld` extension, `pld_test_file.pld`.
Next, open it, and write something like this :

```
PLD_TEST
```

Once you have done this, save the file. In the Project Manager, add `pld_test_file.pld` file by right-clicking the Project Level Defines node.

In the source code write the following :

```
{$IFDEF PLD_TEST}
...
{$ENDIF}
```

There are a number of predefined project level defines. See predefined project level defines

Related topics: Project Manager, Project Settings, Edit Project

Source Files

Source files containing source code should have the extension `.mpas`. The list of source files relevant to the application is stored in project file with extension `.mppav`, along with other project information. You can compile source files only if they are part of the project.

Managing Source Files

Creating new source file

To create a new source file, do the following:

1. Select **File › New Unit** from the drop-down menu, or press Ctrl+N, or click the New File Icon  from the File Toolbar.
2. A new tab will be opened. This is a new source file. Select **File › Save** from the drop-down menu, or press Ctrl+S, or click the Save File Icon  from the File Toolbar and name it as you want.

If you use the New Project Wizard, an empty source file, named after the project with extension `.mpas`, will be created automatically. The mikroPascal PRO for PIC32 does not require you to have a source file named the same as the project, it's just a matter of convenience.

Opening an existing file

1. Select **File › Open** from the drop-down menu, or press Ctrl+O, or click the Open File Icon  from the File Toolbar. In Open Dialog browse to the location of the file that you want to open, select it and click the Open button.
2. The selected file is displayed in its own tab. If the selected file is already open, its current Editor tab will become active.

Printing an open file

1. Make sure that the window containing the file that you want to print is the active window.
2. Select **File › Print** from the drop-down menu, or press Ctrl+P.
3. In the Print Preview Window, set a desired layout of the document and click the OK button. The file will be printed on the selected printer.

Saving file

1. Make sure that the window containing the file that you want to save is the active window.
2. Select **File › Save** from the drop-down menu, or press Ctrl+S, or click the Save File Icon  from the File Toolbar.

Saving file under a different name

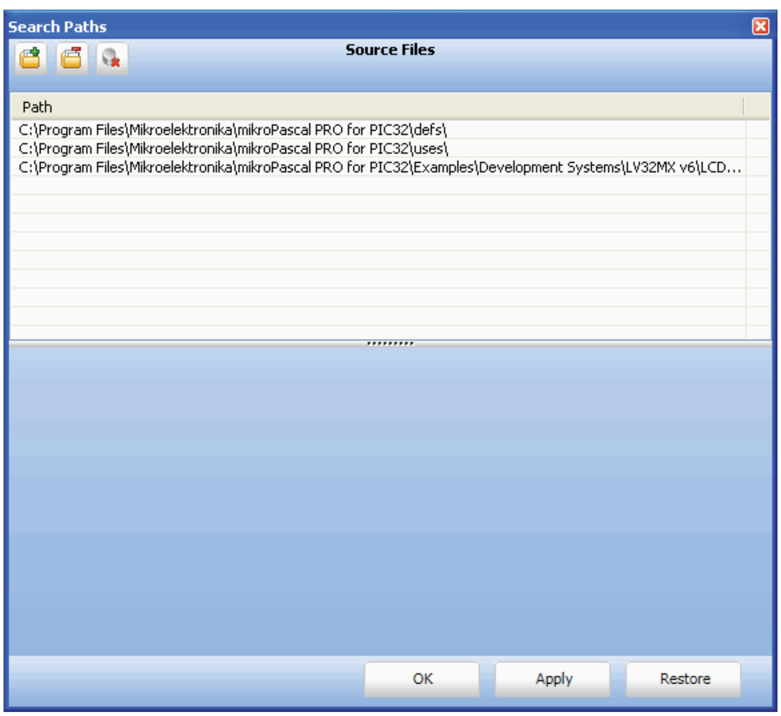
1. Make sure that the window containing the file that you want to save is the active window.
2. Select **File › Save As** from the drop-down menu. The New File Name dialog will be displayed.
3. In the dialog, browse to the folder where you want to save the file.
4. In the File Name field, modify the name of the file you want to save.
5. Click the Save button.

Closing file

1. Make sure that the tab containing the file that you want to close is the active tab.
2. Select **File › Close** from the drop-down menu, or right click the tab of the file that you want to close and select **Close** option from the context menu.
3. If the file has been changed since it was last saved, you will be prompted to save your changes.

Search Paths

You can specify your own custom search paths: select **Project › Edit Search Paths...** option from the drop-down menu:



Following options are available:

Icon	Description
	Add Search Path.
	Remove Search Path.
	Purge Invalid Paths.

Paths for Source Files (`.mpas`)

You can specify either absolute or relative path to the source file. If you specify a relative path, mikroPascal PRO for PIC32 will look for the file in following locations, in this particular order:

1. the project folder (folder which contains the project file `.mpp32`),
2. your custom search paths,
3. mikroPascal PRO for PIC32 installation folder > `Uses` folder.

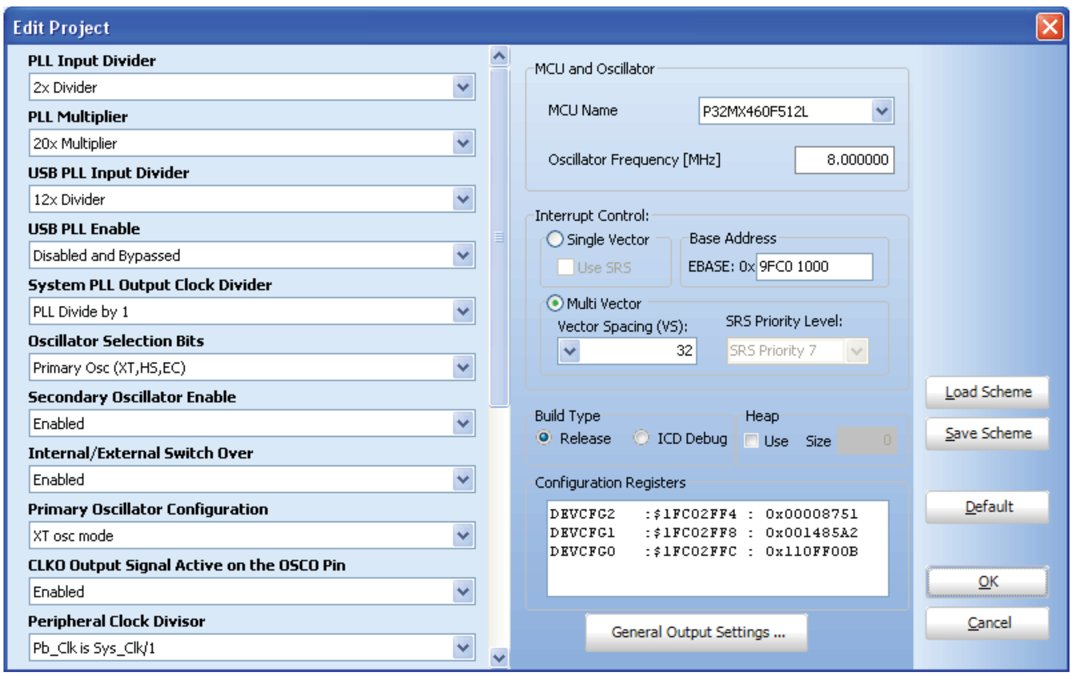
Related topics:File Menu, File Toolbar, Project Manager, Project Settings,

Edit Project

Edit Project gives you option to change MCU you wish to use, change its oscillator frequency and build type. Also, Edit Project enables you to alter specific configuration bits of the selected device.

As you alter these bits, appropriate register values will be updated also. This can be viewed in the **Configuration Registers** pane.
When you have finished configuring your device, you can save bit configuration as a scheme, using Save Scheme button.
In case you need this scheme in another project, you can load it using Load Scheme button.

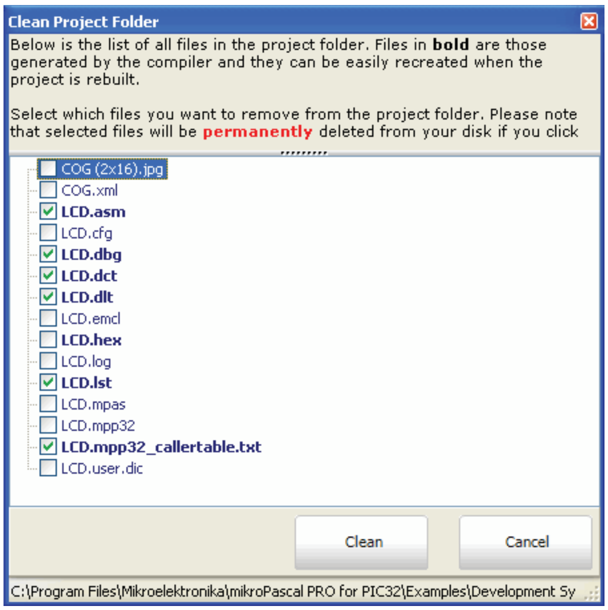
There is also a Default button which lets you select default configuration bit settings for the selected device.



Related topics: Project Settings, Customizing Projects

Clean Project Folder

This menu gives you option to choose which files from your current project you want to delete.
Files marked in bold can be easily recreated by building a project. Other files should be marked for deletion only with a great care, because IDE cannot recover them.



Related topics: Customizing Projects

Compilation

When you have created the project and written the source code, it's time to compile it. Select **Project › Build** from the drop-down menu, or click the Build Icon  from the Build Toolbar. If more more than one project is open you can compile all open projects by selecting **Project › Build All Projects** from the drop-down menu, or click the Build All Projects Icon  from the Build Toolbar.

Progress bar will appear to inform you about the status of compiling. If there are some errors, you will be notified in the Messages Window. If no errors are encountered, the mikroPascal PRO for PIC32 will generate output files.

Output Files

Upon successful compilation, the mikroPascal PRO for PIC32 will generate output files in the project folder (folder which contains the project file `.mpp32`). Output files are summarized in the table below:

Format	Description	File Type
Intel HEX	Intel style hex records. Use this file to program MCU.	`.hex`
Binary	Extended mikro Compiled Library. Binary distribution of application that can be included in other projects.	`.emcl`
List File	Overview of MCU memory allotment: instruction addresses, registers, routines and labels.	`.lst`
Assembler File	Human readable assembly with symbolic names, extracted from the List File.	`.asm`

Assembly View

After compiling the program in mikroPascal PRO for PIC32, you can click the View Assembly icon  or select **Project › View Assembly** from the drop-down menu to review the generated assembly code (`.asm` file) in a new tab window.
Assembly is human-readable with symbolic names.

Related topics: Build Menu, Build Toolbar, Messages Window, Project Manager, Project Settings

Creating New Library

mikroPascal PRO for PIC32 allows you to create your own libraries. In order to create a library in mikroPascal PRO for PIC32 follow the steps bellow:

1. Create a new source file, see Managing Source Files
2. Save the file in one of the subfolders of the compiler's Uses folder:
   `DriveName:\Program Files\Mikroelektronika\mikroPascal PRO for PIC32\Uses\`
3. Write a code for your library and save it.
4. Add `__Lib_Example` file in some project, see Project Manager. Recompile the project.
   If you wish to use this library for all MCUs, then you should go to **Tools › Options › Output settings**, and check **Build all files as library** box.
   This will build libraries in a common form which will work with all MCUs. If this box is not checked, then library will be built for selected MCU.
   Bear in mind that compiler will report an error if a library built for specific MCU is used for another one.
5. Compiled file `__Lib_Example.emcl` should appear in `...\mikroPascal PRO for PIC32\Uses\` folder.
6. Open the definition file for the MCU that you want to use. This file is placed in the compiler's Defs folder:
   `DriveName:\Program Files\Mikroelektronika\mikroPascal PRO for PIC32\Defs\`
   and it is named `MCU_NAME.mlk`, for example `P32MX460F512L.mlk`
7. Add the the following segment of code to `<LIBRARIES>` node of the definition file (definition file is in XML format):
```
<LIB>
  <ALIAS>Example_Library</ALIAS>
  <FILE>__Lib_Example</FILE>
  <TYPE>REGULAR</TYPE>
</LIB>
```
8. Add Library to mlk file for each MCU that you want to use with your library.
9. Click Refresh button in Library Manager
10. `Example_Library` should appear in the Library manager window.

Multiple Library Versions

Library Alias represents unique name that is linked to corresponding Library .emcl file. For example UART library for PIC32MX460F512L is different from UART library for PIC32MX675F512L MCU. Therefore, two different UART Library versions were made, see `mlk` files for these two MCUs. Note that these two libraries have the same Library Alias (UART) in both `mlk` files. This approach enables you to have identical representation of UART library for both MCUs in Library Manager.

Related topics: Library Manager, Project Manager, Managing Source Files

Frequently Asked Questions

This is a list of frequently asked questions about using mikroElektronika compilers.
If your question is not answered on this page, please contact mikroElektronika Support Desk.

Can I use your compilers and programmer on Windows Vista (Windows 7) ?

Our compilers and programmer software are developed to work on and tested on Windows 98, Windows 2000, Windows ME, Windows XP (32 and 64 bit), Windows Vista (32 and 64 bit) and Windows 7 (32 and 64 bit) and they work fine on these operating systems.
You can find the latest drivers on our website.

I am getting “Access is denied” error in Vista, how to solve this problem ?

Please turn off User Account Control (UAC). This should make your software fully functional. To do this, follow the path in your Windows Vista (logged in as administrator) **Control Panel › User Accounts › Turn User Account Control** on or off, uncheck Use User Account Control (UAC) and click OK.

What are differences between mikroC PRO, mikroPascal PRO and mikroBasic PRO compilers ? Why do they have different prices ?

Basically, there is little differences between these compilers. mikroC PRO is standardized with ANSI C, and it is much more complex and it is far more difficult to write the compiler for it. We used a lot more resources for making it than what we used for mikroPascal and mikroBasic. We also worked on some very complex topics such as floating point, typedef, union, a completely new debugger and many other. Because of that there is difference in price.

Why do your PIC compilers don’t support 12F508 and some similar chips ?

Unfortunately our PIC compilers don’t support 12F508 and similar chips because these chips are designed to use 12-bit wide instructions. Our compiler support MCUs which use 14-bit or wider instructions.

What are limitations of demo versions of mikroElektronika’s compilers ?

The only limitation of the free demo version is that it cannot generate hex output over 2K of program words. Although it may sound restrictive, this margin allows you to develop practical, working applications without ever thinking of demo limit. If you intend to develop really complex projects in one of our compilers, you should consider purchasing the license key.

Why do I still get demo limit error when I purchased and installed license key ?

If you are first time installing and registering compiler, you need to follow instructions exactly as described in registration procedure. License is valid only for the computer from which request is made, so license requested from one computer won’t work on another computer. You can find on our site manual and video describing in detail how to get your license. If you previously had an older version of our compiler and have working license key for it but it doesn’t work with new compiler, you have to repeat registration procedure from the new compiler and you will get a new license.

I have bought license for the older version, do I have to pay license for the new version of the compiler ?

No, once you pay for the license key you get a lifetime license. When we release a new major release of the compiler, you might need to repeat registration procedure from your new compiler and you will get new license free of charge.

Do your compilers work on Windows Vista (Windows 7) ?

Yes!

What does this function/procedure/routine do ?

Please see your compiler's Help where all of the functions are explained in detail.

I try to compile one of the provided examples and nothing happens, what is the problem?

You need to open project, not file. When you want to open an example, go to **Project › Open Project**, then browse through projects and choose project file. Now you will be able to compile and program with success.

Can I get your library sources ? I need to provide all sources with my project.

It is our company's policy not to share our source code.

Can I use code I developed in your compilers in commercial purposes ? Are there some limitations ?

Regarding your code, there are no limitations. Your application is your own property and you can do whatever you like with it. If you want to include some of code we provide with our compilers or on our site, you may include them in your project, however, you are not allowed to charge your users for these.

Why does an example provided with your compilers doesn't work ?

All of the examples provided with our compilers are tested and work fine. You need to read commented header of the example and be sure that you have used the same MCU example is written for and that you have hardware connections (DIP switches, jumpers etc.) set as described.

Your example works if I use the same MCU you did, but how to make it work for another MCU ?

You should read your MCU's datasheet. Different MCUs can have different pin assignments and may require different settings. If you need help regarding this, you can find free online books on our website and recommend you starting there. You can also ask for help on our forum.

I need this project finished, can you help me ?

We currently do not do custom projects, however, we can give you some directions when you start working on your project and come to a problem. Also, our forum is very active community and as you can find there experts in different fields, we encourage you to look for help there.

Do you have some discount on your compilers/development systems for students/ professors ?

Since large percentage of our customers are schools, laboratories and students, our prices are already scaled for these kinds of users. If you plan ordering more than one of our products, see special offers page on our website. Also, you can contact our Sales Department and see if you are eligible for some additional discount.

I have a question about your compilers which is not listed here. Where can I find an answer ?

Firstly, look for it in your compiler's Help. If you don't find an answer there, please create a support ticket on our website.